Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Robas

Učinkovito poizvedovanje po povezanih podatkih s porazdelitvijo obremenitve

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Dejan Lavbič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Zahvaljujem se mentorju doc. dr. Dejanu Lavbiču za odlično sodelovanje, strokovno usmerjanje, pomoč in nasvete pri izdelavi tega magistrskega dela.

Zahvaljujem se Mateju Nanutu za podrobno branje magistrskega dela in predloge za izboljšave.

Zahvaljujem se Sonji Grmovšek za odpravo slovničnih napak in njeno dobro voljo pri tem.

Posebna zahvala gre mojim staršem, ki so me brezpogojno podpirali tekom celotnega študija.

Kazalo

Povzetek

Abstract

1 Uvod **1**

2 Povezani podatki **5**

2.1 Nabor tehnologij 6

3 Delci povezanih podatkov **9**

3.1 Delci vzorcev trojčkov 11

3.2 Uporaba relacijske baze za hrambo trojčkov 20

4 Zmanjšanje števila vsebnostnih poizvedb s pomočjo metapodatkov **25**

4.1 Bloomov filter 26

4.2 Razširitev AMF 27

5 Opis naše razširitve za zmanjšanje števila vsebnostnih poizvedb **31**

5.1 Različice naše razširitve 32

5.2 Podroben opis algoritmov 33

6 Empirične meritve **41**

6.1 Metodologija 41

6.2 Rezultati meritev 52
6.3 Rezultati meritev pri razširitvi z omejitvijo predikatov 71

7 Grafični vmesnik za sestavljanje in izvajanje poizvedb 83

8 Zaključek 89
8.1 Rezultati našega dela 89
8.2 Nadaljnje delo 92
8.3 Sklep 93

Literatura 99

Seznam uporabljenih kratic

kratica	angleško	slovensko
AMF	Approximate Membership Functions	Funkcije približnega vsebovanja
API	Application Programming Interface	Aplikacijski programski vmesnik
BGP	Basic Graph Pattern	Osnovni vzorec grafa
CPE	Central Processing Unit	Centralno procesna enota
DB	Database	Podatkovna baza
GCS	Golomb Compressed Sets	Množice, stisnjene z Golombovim kodiranjem
HDT	Header, Dictionary, Triples	Glava, imenik, trojčki
HTML	HyperText Markup Language	Jezik za označevanje nadbesedila
HTTP	HyperText Transfer Protocol	Protokol za prenašanje nadbesedila
PID	Process identifier	Identifikator procesa
RDF	Resource Description Framework	Ogrodje za opis virov
REST	Representational state transfer	Predstavitveni prenos stanja
SOAP	Simple Object Access Protocol	Preprost dostopni protokol za objekte

kratica	angleško	slovensko
SPARQL	**SPARQL Protocol and RDF Query Language**	**Protokol SPARQL in poizvedovalni jezik RDF**
TPF	**Triple Pattern Fragment**	**Delec vzorca trojčka**
URI	**Uniform Resource Identifier**	**Enolični identifikator vira**
URL	**Uniform Resource Locator**	**Enolični krajevnik vira**

Povzetek

Naslov: Učinkovito poizvedovanje po povezanih podatkih s porazdelitvijo obremenitve

Na spletu so podatki predstavljeni na različne načine v različnih oblikah in zato med sabo niso združljivi. Isti problem se pojavi tudi pri spletnih programskih vmesnikih, saj je za povezovanje z zunanjimi podatki ponavadi potrebno narediti odjemalca, prilagojenega za dane podatke. Ta problem rešujejo povezani podatki. Uporabo povezanih podatkov zavirajo problemi s performancami poizvedovanja in posledično z dostopnostjo oddaljenih točk SPARQL. Z delci vzorcev trojčkov lahko pri poizvedovanju del obremenitve prestavimo na odjemalca, vendar pri tem prenesemo večjo količino podatkov. Obstoječa razširitev AMF zmanjša število zahtevkov in s tem posledično tudi količino podatkov pri določenih poizvedbah, medtem ko pri ostalih poizvedbah količino podatkov poveča. V tem delu predstavljamo našo razširitev, s katero smo poskusili zmanjšati tako število zahtevkov kot količino prenesenih podatkov tako, da smo v metapodatke dodali Bloomov filter, ki vsebuje trojčke, povezane z rezultati na prikazani strani delca vzorca trojčka. Našo razširitev smo primerjali z osnovno različico in z različico AMF ter pri našem testu dosegli dobre rezultate. Prispevali smo tudi popravek razširitve AMF, ki je že vključen v uradni repozitorij, in za konec razvili preprost grafični vmesnik, ki omogoča sestavljanje poizvedb SPARQL in njihovo izvajanje z uporabo naše razširitve.

Ključne besede: povezani podatki, SPARQL, delci vzorca trojčkov, procesiranje na odjemalcu, spletne tehnologije.

Abstract

Title: Efficient querying of Linked Data by distributing workload

Online data is presented in different ways and in various forms which are not mutually compatible. This problem is also present in Web APIs, because we usually have to implement a specialised client, suited for the kind of data the Web service is providing. This problem is solved with Linked Data. The problem with Linked Data is the query performance and the availability of remote SPARQL endpoints. With Triple Pattern Fragments we can execute SPARQL queries by transferring some workload to the client, but in contrast we have to transfer more data. The existing AMF extension reduces the amount of HTTP requests and consequently the amount of transferred data on some queries, while increasing the amount of transferred data with others. In this thesis we present our extension, where we try to lower the amount of HTTP requests and the amount of transferred data by extending the metadata with a Bloom filter, containing data, linked with triples on the current page of the Triple Pattern Fragment. We have compared our extension with the AMF extension and achieved encouraging results. We have also proposed a fix for the AMF extension, which is already included in the official repository. Finally, we have developed a simple graphical user interface that enables composition of SPARQL queries and their execution using our extension.

Keywords: Linked Data, SPARQL, triple pattern fragments, client-side processing, Web technologies.

Poglavje 1

Uvod

Uporabniki se vedno bolj zanašamo na pridobivanje podatkov preko svetovnega spleta s pomočjo najrazličnejših naprav, kot so računalniki, telefoni, tablični računalniki in navsezadnje nosljivi računalniki (angl. *wearable computers*). Na spletu se nahaja velika količina podatkov, ki je predstavljena v različnih oblikah, in jih med sabo težko povezujemo. Lahko so predstavljeni npr. v tabeli v dokumentu HTML, ponujeni v datoteki kot vrednosti, ločene z vejicami (angl. *comma separated values*) itd. Preprost primer je seznam poštnih številk Pošte Slovenije [9], kjer imamo možnost pregledovanja seznama kot dokumenta HTML, obenem pa imamo možnost prenosa datoteke `xlsx` z enakovrednimi podatki. Druga vrsta ponujanja podatkov so spletni programski vmesniki (angl. *Web APIs*), za katere se ponavadi uporablja bodisi bolj zapleten protokol SOAP, ki sicer ponuja kup dodatnih funkcij (npr. protokoli za zaščito [32]), bodisi preprostejša arhitektura REST, na kateri gradijo programske vmesnike spletna podjetja z velikim tržnim deležem, kot sta npr. Google in Amazon [35]. Če hočemo narediti aplikacijo, ki uporablja te spletne vmesnike, imamo lahko precej dela z usklajevanjem naših podatkov s podatki, ki jih dobimo preko njih. Kot omenja članek [27], moramo v naši aplikaciji za vsak spletni vmesnik posebej napisati kodo za odjemalca, kar zahteva znaten napor. Podatki, ki jih ponujajo taki spletni vmesniki, niso na voljo spletnim iskalnikom oziroma splošnim spletnim agentom. Ta problem

bi lahko rešili z objavo podatkov v obliki povezanih podatkov, kjer so podatki med sabo povezani kot dokumenti HTML, s to razliko, da so povezave tipizirane. Članek [15] govori o preprosti skripti, ki zbira podatke iz spletnih vmesnikov in jih zunanjim sistemom ponuja kot povezane podatke. Bolj učinkovito bi seveda bilo, da bi ti spletni vmesniki že sami ponujali svoje podatke v obliki povezanih podatkov. V tem primeru se ne bi bilo več potrebno obremenjevati s prilagajanjem na različne spletne programske vmesnike do te mere, da bi bilo dodajanje novega vira podatkov lahko avtomatizirano.

Problem, ki se pri povezanih podatkih pojavlja in ga opisuje članek [13], so performance pri poizvedovanju in posledično dostopnost oddaljenih točk SPARQL. To močno zavira njihovo uporabo. Zaradi tega je v članku [45] predlagan način objave podatkov, imenovan delci vzorcev trojčkov (angl. *triple pattern fragments*). Z delci vzorcev trojčkov je predlagan nov način poizvedovanja po povezanih podatkih, tako da so le-ti objavljeni v obliki manjših delcev, odjemalec pa omogoča izvajanje poizvedb SPARQL, s tem da ob izvedbi posamezne poizvedbe prenese le tiste delce, ki jih potrebuje. Tako je obremenitev porazdeljena med strežnikom in odjemalcem, hkrati pa je predpomnjenje bolj učinkovito, saj se pogosto zgodi, da različne poizvedbe pri izvajanju potrebujejo iste delce. Problem pri tem je število zahtevkov in količina podatkov, potrebna za izvajanje poizvedb. Ta problem s pomočjo verjetnostne podatkovne strukture poskuša omiliti razširitev *AMF*, opisana v članku [40]. V tem delu na malo drugačen način poskušamo doseči podobno oziroma večje izboljšanje, predvsem glede količine prenesenih podatkov pri izvajanju poizvedb. Ob raziskovanju nam uspe tudi odkriti in odpraviti napako pri razširitvi *AMF*, s čimer občutno izboljšamo performance. Popravek, ki je na kratko opisan v razdelku 4.2, je bil kmalu vključen v javni repozitorij. Poleg tega za delce vzorcev trojčkov naredimo še vmesnik, ki omogoča uporabo relacijske podatkovne baze MariaDB kot izvor povezanih podatkov ter za zaključek celote zasnujemo še preprost grafični vmesnik, ki preko spletnega brskalnika omogoča sestavljanje poizvedb SPARQL in njihovo izvajanje z uporabo naše razširitve.

V poglavju 2 govorimo o glavnih principih in uporabljenih tehnologijah pri povezanih podatkih. V poglavju 3 predstavimo delce vzorcev trojčkov in način izvajanja poizvedb SPARQL nad njimi, nato pa še naš vmesnik, ki omogoča uporabo relacijske baze kot podatkovnega vira. V poglavju 4 na kratko predstavimo Bloomove filtre in razširitev *AMF*. V poglavju 5 podrobno predstavimo našo razširitev in njene različice. V poglavju 6 predstavimo in opišemo meritve, v katere so, poleg različic z našimi razširitvami, vključene še osnovna različica, različica z razširitvijo *AMF* pred našim popravkom ter različica z razširitvijo *AMF* po našem popravku. V poglavju 7 na kratko predstavimo naš grafični vmesnik, ki omogoča sestavljanje preprostih poizvedb in njihovo izvajanje z uporabo naše razširitve. V poglavju 8 povzamemo rezultate našega dela, podamo nekaj smernic za nadaljnje delo ter naredimo sklep.

Poglavje 2

Povezani podatki

Pojem povezani podatki (angl. *Linked Data*) se nanaša na nabor najboljših praks objavljanja in povezovanja strukturiranih podatkov na spletu. V splošnem gre pri povezanih podatkih za tipizirane povezave med podatki iz različnih virov. Ti viri so lahko bodisi znotraj iste organizacije bodisi porazdeljeni med več različnih organizacij. Povezani podatki so objavljeni na spletu na tak način, da so strojno berljivi in jih je možno naknadno povezovati z ostalimi viri. Gre za povezovanje podatkov na podoben način, kot so na spletu povezani dokumenti HTML. Medtem ko so dokumenti HTML povezani z netipiziranimi povezavami, povezani podatki temeljijo na formatu RDF, kjer so povezave med objekti tipizirane. Rezultat je splet podatkov oziroma bolj točno splet objektov in pojmov, ki so predstavljeni s podatki na spletu [14].

Osnovni principi povezanih podatkov so [6]:

1. Uporabi URI-je kot imena za stvari (tako za objekte kot pojme).
2. Uporabi HTTP URI-je, tako, da lahko ljudje pogledajo ta imena (npr. preko spletnega brskalnika).
3. Ko nekdo pogleda URI, ponudi koristne informacije z uporabo standardov (RDF, SPARQL).
4. Vključi povezave do drugih URI-jev, da lahko odkrijejo več stvari.

Ti principi se skladajo s principi in standardi spleta.

2.1 Nabor tehnologij

Povezani podatki temeljijo na temeljnih spletnih tehnologijah: URI (angl. *Uniform Resource Identifier*) in HTTP (angl. *HyperText Transfer Protocol*). Medtem ko z URL-ji (angl. *Uniform Resource Locators*) naslavljamo posamezne dokumente in ostale vire na spletu, URI-ji omogočajo bolj splošno naslavljanje poljubnih entitet na svetu. Če se URI-ji začnejo s `http://`, potem jih lahko kot URL-je zahtevamo preko protokola HTTP in o njih pridobimo dodatne informacije. Čeprav URI-ji lahko predstavljajo pojme ali fizične stvari, ki jih ne moremo prenesti preko spleta, lahko o njih pridobimo opise, dodatne informacije in povezane podatke.

Dve omenjeni tehnologiji sta dopolnjeni z RDF, ki definira podatkovni model v obliki grafa, v katerem povezujemo entitete s tipiziranimi povezavami. Podatki so zapisani v obliki trojčkov oblike *subjekt, predikat, objekt*. Tako *subjekt* kot *objekt* sta URI-ja, ki predstavljata vir oziroma v primeru *objekta* opcijski niz znakov (literal, ki lahko predstavlja bodisi niz znakov bodisi število ...). Predikat je tudi predstavljen z URI in pove, v kakšnem razmerju sta *subjekt* in *objekt*.

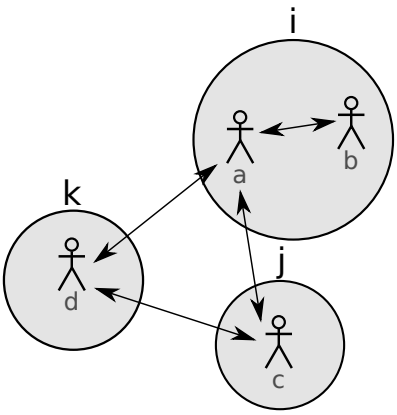
Če imamo npr. predikat `http://xmlns.com/foaf/0.1/knows` in je prva oseba (*subjekt*) A s tem predikatom povezana z drugo osebo (*objekt*) B, si to razlagamo kot: oseba A pozna osebo B. V modelu RDF je to trojček `(A http://xmlns.com/foaf/0.1/knows B)`, pri čemer sta A in B URI-ja, kjer vsak predstavlja svojo osebo. Dva vira (v našem primeru osebi A in B) sta lahko izvzeta iz dveh različnih podatkovnih virov na spletu (npr. iz dveh družbenih omrežij). Ti podatki so med sabo združljivi. Trojčke RDF si zato lahko predstavljamo na podoben način, kot spletne povezave med stranmi HTML.

Na sliki 2.1 je ponazorjen graf, ki opisuje poznanstva, spletena med tremi različnimi družbenimi omrežji. Ostali podatki uporabnika (polno ime, rojstni datum, interesi, objave ...) so za vsakega uporabnika shranjeni v njegovem omrežju. Model RDF, kot smo ga opisali, je lahko zapisan z različnimi sintaksami. Primer zapisa omenjenih poznanstev v obliki Turtle [20] je viden na zapisu 2.1.

```
@prefix foaf:   <http://xmlns.com/foaf/0.1/> .
@prefix oi:     <http://omrezje-i.si/oseba/> .
@prefix oj:     <http://omrezje-j.si/oseba/> .
@prefix ok:     <http://omrezje-k.si/oseba/> .

oi:a foaf:knows oi:b .
oi:b foaf:knows oi:a .
oi:a foaf:knows ok:d .
oi:a foaf:knows oj:c .
ok:d foaf:knows oi:a .
ok:d foaf:knows oj:c .
oj:c foaf:knows ok:d .
oj:c foaf:knows oi:a .
```

Zapis 2.1: Trojčki, zapisani v obliki Turtle, s katerimi lahko opišemo graf na sliki 2.1.



Slika 2.1: Graf poznanstev.

Poglavje 3

Delci povezanih podatkov

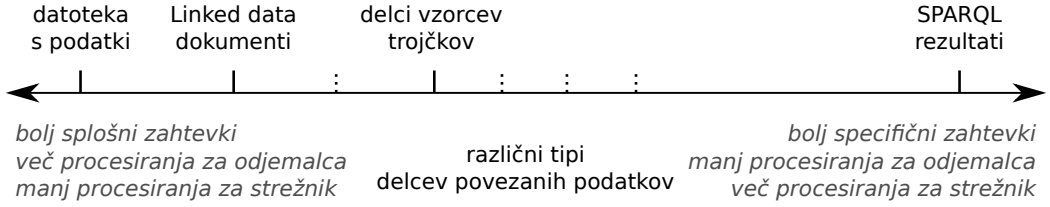
Članek [13] govori o problemu dostopnosti oddaljenih točk SPARQL, saj je samo okoli tretjina le-teh dostopna 99 % časa. Ob koncu leta 2013 je bila povprečna dostopna točka SPARQL nedosegljiva več kot dan in pol *vsak mesec*. Poleg tega obstajajo velike razlike v učinkovitosti poizvedovanja (3–4 velikostnih redov). To pomeni, da ne moremo razviti zanesljive aplikacije, ki bi se zanašala na poizvedovanje preko tovrstnih dostopnih točk. Zaradi tega je razvoj aplikacij semantičnega spleta močno zaviran.

Ena rešitev tega problema je, da ponudimo uporabnikom statične datoteke s podatki RDF. Problem ponujanja podatkov RDF v statičnih datotekah je, da so podatkovne zbirke lahko zelo velike, uporabnik pa morda potrebuje samo majhen del podatkov. DBpedia ima npr. več kot 100 milijonov trojčkov [26]. Poleg tega mora imeti uporabnik pri sebi programsko opremo, ki mu omogoča poizvedovanje. Pri tem načinu moramo tudi skrbeti za nenehno posodabljanje velike količine podatkov. Vse to nasprotuje viziji semantičnega spleta oziroma spleta na splošno. Tak način dostopanja do povezanih podatkov ni primeren za mobilne naprave, saj imajo le-te strožje omejitve prenosa podatkov in računskih zmogljivosti. Problem je omenjen tudi v članku [28], kjer je opisana mobilna aplikacija, ki dela s povezanimi podatki tako, da del podatkov DBpedije v obliki HDT prenese na mobilno napravo in na sami napravi izvaja poizvedbe. Ta ideja je skladna z idejo delcev povezanih po-

datkov, le da je slednja natančneje definirana.

Osnovna ideja delcev povezanih podatkov (angl. *Linked Data Fragments*), kot jo opisuje članek [44], je, da so na dostopni točki objavljeni delci (angl. *fragments*) celotnega nabora povezanih podatkov (v obliki trojčkov), ki so razdeljeni po straneh, pri čemer lahko trojčke filtriramo z izbirnikom (angl. *selector*). V skladu z idejo povezanih podatkov so pri delcih povezanih podatkov prisotni metapodatki z dodatnimi informacijami o podatkih ter, kot jih imenuje omenjeni članek, kontrole (angl. *controls*), ki vsebujejo povezave do ostalih strani rezultatov in do ostalih delcev povezanih podatkov.

Tako odgovore na poizvedbe SPARQL kot statične datoteke s podatki RDF lahko obravnavamo kot delce povezanih podatkov. Slika 3.1 prikazuje različne načine pridobivanja podatkov s perspektive delcev povezanih podatkov. Označena sta dva skrajna pristopa — na eni strani dostop do datoteke s podatki, na drugi pa dostop do oddaljene dostopne točke SPARQL. V prvem primeru strežnik ponuja le statično datoteko, ki jo lahko uporabnik prenese na svoj računalnik in pri sebi izvaja poizvedbe, v drugem primeru pa poizvedovanje opravlja strežnik, uporabnik pa dobi le specifične podatke, ki so v odgovoru poizvedbe. Izbirnik je v primeru statične datoteke s podatki zelo splošen (URL do datoteke, ki vsebuje želene podatke), v primeru dostopne točke SPARQL pa zelo natančen (sama poizvedba).



Slika 3.1: Vse spletne vmesnike, ki ponujajo trojčke RDF, si lahko predstavljamo kot ponudnike delcev vzorcev trojčkov (povzeto po članku [45]).

3.1 Delci vzorcev trojčkov

Delci vzorcev trojčkov so delci povezanih podatkov, s pomočjo katerih lahko izvajamo poizvedbe SPARQL na tak način, da je delo porazdeljeno med ponudnikom delcev povezanih podatkov (strežnikom) in odjemalcem.

3.1.1 Definicije osnovnih pojmov

V tem razdelku so predstavljene definicije osnovnih pojmov, povzete iz člankov [44] in [45], ki jih uporabljamo tudi v opisu našega pristopa.

Definicija 1. Trojčki RDF. Če vzamemo $\mathcal{U}$, $\mathcal{B}$, $\mathcal{L}$ in $\mathcal{V}$ kot množice vseh URI-jev, praznih vozlišč, literalov in spremenljivk, potem je $\mathcal{T} = (\mathcal{U} \cup \mathcal{B}) \times (\mathcal{U} \cup \mathcal{B}) \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ množica vseh trojčkov RDF. Zaradi poenostavitve v nadaljnjem besedilu ne bomo obravnavali praznih vozlišč.

Definicija 2. Vzorec trojčka. Vsak $tp \in (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V}) \times (\mathcal{U} \cup \mathcal{V} \cup \mathcal{L})$ predstavlja vzorec trojčka. Vsaka končna množica vzorcev trojčkov je BGP (angl. *basic graph pattern*).

Definicija 3. Poizvedba SPARQL. Poizvedba SPARQL $\mathcal{P}$ je sestavljena iz BGP-jev, ki jih lahko sestavljamo z različnimi operatorji. Rezultat poizvedbe SPARQL nad množico trojčkov RDF $\mathcal{G} \subseteq \mathcal{T}$ je v obliki množice preslikav $p : \mathcal{V} \to (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$, ki predstavljajo preslikave do rešitve (angl. *solution mappings*).

Definicija 4. Rezultati vzorca trojčka. Če imamo množico trojčkov $\mathcal{G} \subseteq \mathcal{T}$ in podan vzorec trojčka tp, potem je rezultat vzorca trojčka množica vseh trojčkov $\mathcal{R}_{tp} \subseteq \mathcal{G}$, pri čemer velja $\forall t \in \mathcal{R}_{tp} \exists p(t = p[tp] \wedge p[tp] \in \mathcal{G})$. Pri tem $p[tp]$ predstavlja trojček, pridobljen z uporabo preslikave p nad tp.

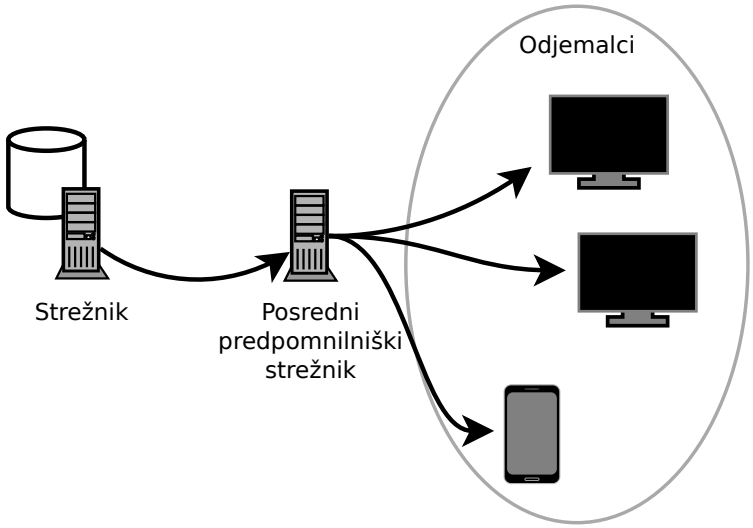
Definicija 5. Vsebnostna poizvedba. Vsebnostna poizvedba nad množico trojčkov $\mathcal{G}$ je poizvedba za rezultate vzorca trojčka tp, kjer tp nima spremenljivk (torej je tp trojček): $tp \in \mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$. Zanima nas, če velja $tp \in \mathcal{G}$. Gre torej za preverjanje, če trojček tp obstaja v $\mathcal{G}$.

3.1.2 Strežnik

Pri delcih vzorcev trojčkov je izbirnik (angl. *selector*) vzorec trojčka. Strežnik ponuja rezultate vzorca trojčka, razdeljene po straneh. Delec vzorca trojčka, ki ga vrača strežnik, vsebuje:

- **podatke:** rezultate vzorca trojčka (na trenutni strani),
- **metapodatke:** približno število vseh trojčkov v rezultatih (po vseh straneh),
- **kontrole:** povezave (URL) do ostalih strani rezultatov podanega vzorca trojčka in do ostalih delcev vzorcev trojčkov (korenski URL do delca z vsemi trojčki v bazi).

Ker je izbirnik namesto specifične poizvedbe SPARQL v vsakem primeru le vzorec trojčka, se pogosto zgodi, da se pri večih različnih poizvedbah tekom izvajanja pojavi isti izbirnik. Če je vsaj en vzorec trojčka v BGP-ju skupen večim poizvedbam (kjer ime spremenljivke ni pomembno), pride do enakih zahtevkov. Zaradi tega dejstva je smiselno uporabiti posredniški predpomnilniški strežnik (angl. *reverse proxy server*), ki posreduje zahteve od strežnika do uporabnika, pri čemer si odgovore strežnika shranjuje v predpomnilnik, da jih potem ob enakih zahtevah lahko vrne odjemalcu brez obremenjevanja strežnika delcev vzorcev trojčkov. Taka topologija je predstavljena z diagramom na sliki 3.2. Navodila za predpomnjenje je možno določiti tako, da v zaglavje (angl. *header*) odgovora strežnika dodamo polje `Cache-Control`. Čas veljavnosti posamezne strani se npr. nastavi tako, da v zaglavje dodamo `Cache-Control: max-age=t`, kjer je t število sekund [22]. Predpomnjenje se lahko dogaja tudi pri samih odjemalcih, če imajo to možnost (npr. predpomnilnik spletnega brskalnika). Spletni brskalnik prav tako upošteva navodila za predpomnjenje, ki so v zaglavju. Navodila lahko predpomnilniški posredniški strežnik ob posredovanju prepiše tako, da predpomnjenje na odjemalcu deluje drugače.



Slika 3.2: Diagram topologije omrežja s posrednim predpomnilniškim strežnikom.

3.1.3 Odjemalec

Odjemalec omogoča poizvedovanje po podatkih s pomočjo poizvedovalnega jezika SPARQL. Ker strežnik nudi dostop do delcev vzorcev trojčkov, mora odjemalec pridobiti ustrezne podatke s pomočjo le-teh. Zaželeno je, da odjemalec vsled izvedbe poizvedbe SPARQL naredi čim manj zahtev HTTP. Odjemalec uporablja metapodatke, konkretno število strani, za planiranje poizvedbe.

BGP-ji so glavni gradniki poizvedb SPARQL. Ostali operatorji, ki se nahajajo med BGP-ji, se izvajajo na strani odjemalca in niso specifični za predstavljen način pridobivanja podatkov. Podpora teh je omejena in ni v obsegu pričujočega dela. Visokonivojski algoritem za izvedbo BGP-jev povzamemo z naslednjimi koraki, opisanimi v članku [44]:

1. Za vsak vzorec trojčka tp_i v BGP $B = \{tp_1, ..., tp_n\}$ pridobi prvo stran ϕ_1^i delca f_i danega vzorca trojčka tp_i, ki vsebuje število vsebovanih trojčkov cnt_i. Izberi tak ϵ, da velja $cnt_\epsilon = min(\{cnt_1, ..., cnt_n\})$. Če je $cnt_\epsilon = 0$, vrni $\{\emptyset\}$.

2. Pridobi vse ostale strani izbranega delca z najmanjšim številom vsebovanih trojčkov f_ϵ. Za vsak trojček t v celotnem delcu ustvari preslikavo do rešitve p_t, tako da je $p_t[tp_\epsilon] = t$ (glej definicijo 4). Nato sestavi nov BGP $B_t = \{tp \mid tp = p_t[tp_j] \wedge tp_j \in B\} \setminus \{t\}$. Če je $B_t \neq \emptyset$, najdi preslikave r_{B_t} z rekurzivnim klicanjem algoritma nad B_t, sicer velja $r_{B_t} = \{p_\emptyset\}$, pri čemer je $p_\emptyset$ prazna preslikava.

3. Vrni vse preslikave do rešitve $p \in \{p_t \cup p' \mid p' \in r_{B_t}\}$.

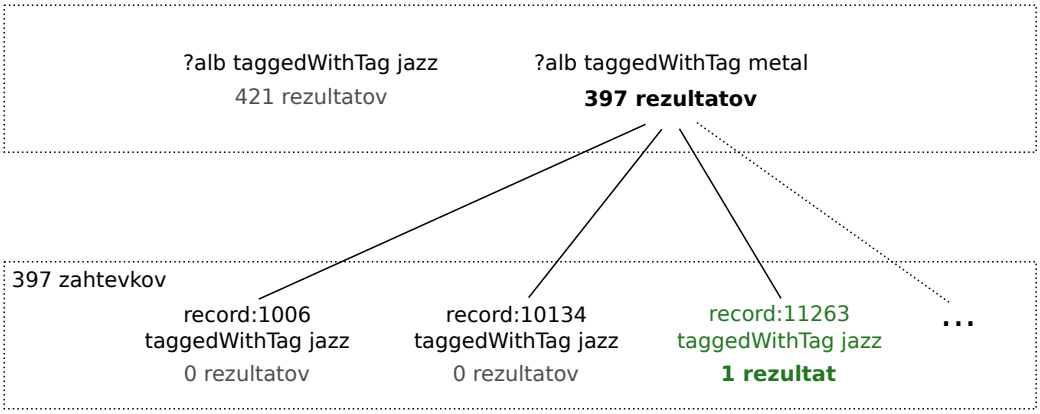
Z rekurzivnim klicanjem predstavljenega algoritma, kjer je vsakič izbran delec, ki vsebuje najmanj trojčkov, je zmanjšano število potrebnih zahtevkov do strežnika. Zaradi poenostavitve v zgornjem postopku ni opisano, da v primeru, ko naletimo na delec, ki ne vsebuje nobenega trojčka, zavržemo dotični niz preslikav, saj ta ne vodi do rezultata.

Algoritmi, ki se izvajajo na odjemalcu pri poizvedovanju, so realizirani s t. i. iteratorji. Iterator BGP in iterator vzorca trojčka sta v tem razdelku predstavljena s psevdokodo (algoritem 1 in 2).

Primeri, ki jih omenjamo, se nanašajo na zbirko RDF, ustvarjeno iz podatkov obsežnega repozitorija proste glasbe Jamendo [2]; v nadaljnjem besedilu jo imenujemo preprosto "Jamendo". V tej zbirki imamo albume, ki so preko predikata `tags:taggedWithTag` povezani z oznakami, ki predstavljajo različne glasbene žanre, kot npr. `tag:jazz`, `tag:rock` in `tag:metal`. Zaradi lažje berljivosti smo URI-je zapisali v obliki Turtle [20]. Predpona `tag:` predstavlja del URI-ja `http://dbtune.org/jamendo/tag/`, predpona `tags:` pa del URI-ja `http://www.holygoat.co.uk/owl/red-wood/0.1/tags/`. Slednja definicija je zapisana v prvih dveh vrsticah pri poizvedbi 3.1.

Za primer si poglejmo preprosto poizvedbo 3.1, ki išče po subjektih, označenih tako z oznako `tag:jazz` kot z oznako `tag:metal`. V tem primeru imamo vzorca trojčkov tp_1 = `(?alb tags:taggedWithTag tag:jazz)` in tp_2 = `(?alb tags:taggedWithTag tag:metal)` ter BGP $B = \{tp_1, tp_2\}$. V množici trojčkov $\mathcal{G}$ torej iščemo tak *subjekt* (ker je spremenljivka del subjekta) trojčka $t \in \mathcal{G}$, da je trojček $t \in tp_1 \wedge t \in tp_2$. Vzorec

trojčka tp_1 ima 421 rezultatov, vzorec trojčka tp_2 pa ima 397 rezultatov. Nazoren prikaz opisane izvedbe je prikazan na sliki 3.3. Ker ima vzorec trojčka tp_2 manj rezultatov kot tp_1, iterator BGP (algoritem 1) izbere rezultate vzorca tp_2, nato pa iterator vzorca trojčka (algoritem 2) *za vsak rezultat* vzorca trojčka tp_2 veže *objekt* na spremenljivko `?alb` in z vzorcem trojčka tp_1 po vezavi naredi temu primerno vsebnostno poizvedbo. V tem primeru je vsebnostnih poizvedb 397 (število rezultatov vzorca trojčka tp_2).



Slika 3.3: Prikaz izvedbe poizvedbe 3.1.

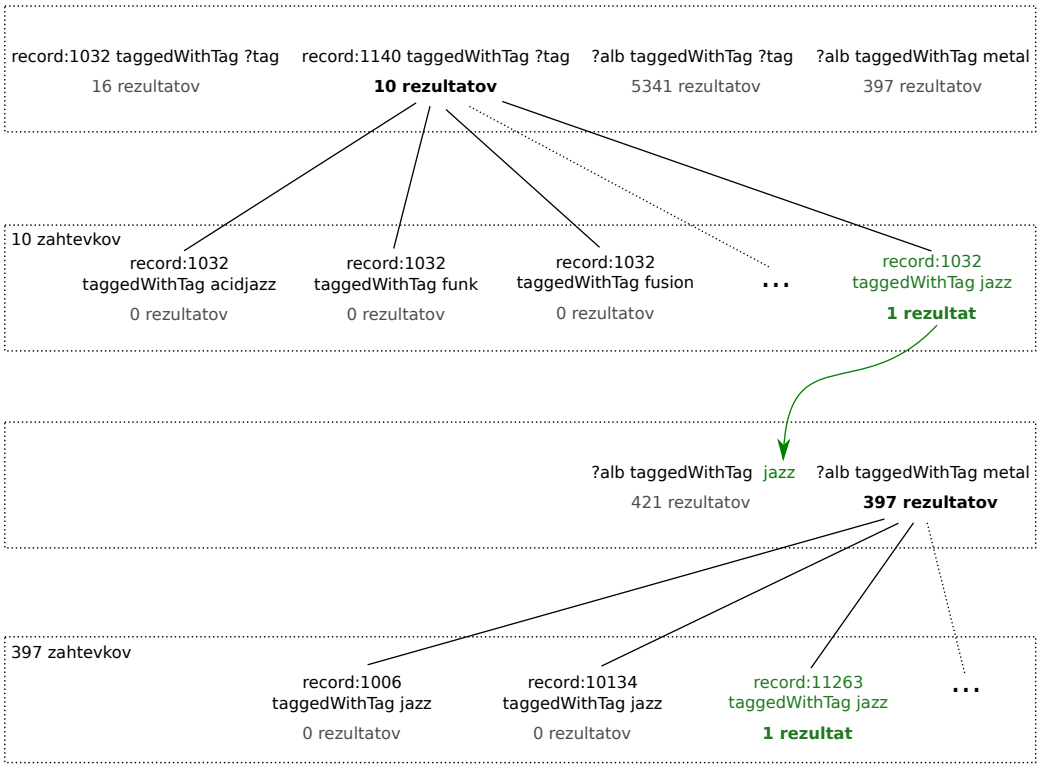
```
PREFIX tags: <http://www.holygoat.co.uk/owl/redwood/0.1/tags/>
PREFIX tag:  <http://dbtune.org/jamendo/tag/>
SELECT ?alb WHERE {
    ?alb tags:taggedWithTag tag:jazz.   # 421
    ?alb tags:taggedWithTag tag:metal.  # 397
}
```

Poizvedba 3.1: Poizvedba SPARQL, za katero odjemalec TPF naredi 397 vsebnostnih poizvedb.

```
PREFIX tags:   <http://www.holygoat.co.uk/owl/redwood/0.1/tags/>
PREFIX tag:    <http://dbtune.org/jamendo/tag/>
PREFIX record: <http://dbtune.org/jamendo/record/>

SELECT ?alb WHERE {
    record:1032 tags:taggedWithTag ?tag.     # 16
    record:1140 tags:taggedWithTag ?tag.     # 10
    ?alb tags:taggedWithTag ?tag.            # 5.341
    ?alb tags:taggedWithTag tag:metal.       # 397
}
```

Poizvedba 3.2: Poizvedba SPARQL, za katero odjemalec TPF naredi 407 vsebnostnih poizvedb.



Slika 3.4: Prikaz izvedbe poizvedbe 3.2. Namesto celotnih URI-jev so zaradi berljivosti prikazane zadnje besede le-teh.

Drug primer je poizvedba 3.2, ki išče oznako `?tag`, s katero sta označena tako `record:1032` kot `record:1140` in nato vse *subjekte*, ki nosijo tako oznako `?tag` kot oznako `tag:metal`.

V tem primeru imamo vzorce trojčkov:

- tp_1 = `(record:1032 tags:taggedWithTag ?tag)` (16 rezultatov),
- tp_2 = `(record:1140 tags:taggedWithTag ?tag)` (10 rezultatov),
- tp_3 = `(?alb tags:taggedWithTag ?tag)` (5.341 rezultatov) in
- tp_4 = `(?alb tags:taggedWithTag tag:metal)` (397 rezultatov)

ter BGP $B = \{tp_1, tp_2, tp_3, tp_4\}$. Tako kot pri prejšnji poizvedbi, iterator BGP (algoritem 1) izbere vzorec trojčka, ki ima najmanj rezultatov, torej tp_2. Nato iterator vzorca trojčka (algoritem 2) za vsak rezultat veže *subjekt* trojčka na spremenljivko `?tag` in z naslednjim vzorcem trojčka, ki ima najmanj rezultatov (tp_1) po vezavi naredi vsebnostno poizvedbo. Na tem mestu je narejenih 10 vsebnostnih poizvedb. Ker je rezultat samo en (`?tag` se preslika v `tag:jazz`), je B na tem mestu enak kot na začetku izvajanja poizvedbe 3.1, ki smo jo obravnavali pri prvem scenariju. Od tu naprej je postopek enak kot pri prvem scenariju. Prikaz izvedbe celotne poizvedbe 3.2 je prikazan na sliki 3.4. V tem primeru je vsebnostnih poizvedb 407, torej 10 več kot pri prvem scenariju. Posebnost uporabe omenjenih iteratorjev je, da se plan izvedbe določa sproti. Če bi npr. po izvedbi prvega dela (tp_1 in tp_2) dobili preslikavo iz `?tag` v `tag:fusion`, bi iterator BGP namesto vzorca trojčka tp_4 izbral tp_3, saj bi tako v drugem delu naredil manj vsebnostnih poizvedb. Gre za dinamično gradnjo drevesa iteratorjev ob izvajanju poizvedbe.

Algoritem 1 Odjemalec - iterator BGP

Podatki:

I_p = prednik ▷ Starševski iterator.
$B = \{tp_1, ..., tp_n\}$ ▷ $n \geq 2$, sicer začnemo z iteratorjem vzorca trojčka.
ϕ_0 = začetna stran
$I = nil$

1: **function** BasicGraphPatternIterator.next()
2: $p \leftarrow nil$
3: **while** $p = nil$ **do**
4: **while** $I = nil$ **do**
5: $p_p \leftarrow I_p.next()$
6: **if** $p_p = nil$ **then**
7: **return** nil
8: **end if**
9: $\Phi \leftarrow \{\phi_1^i \mid \phi_1^i =$ pridobi prvo stran delca $p_p[tp_i]\}$
10: ▷ Pridobimo prvo stran za vsak vzorec trojčka.
11:
12: $\epsilon \leftarrow i$, tako da je $stRez_{\phi_1^i} = min(\{stRez_{\phi_1^1}, ..., stRez_{\phi_1^n}\})$
13: ▷ "$stRez_{\phi_1^n}$" predstavlja število rezultatov n-tega vzorca trojčkov.
14:
15: $I_\epsilon \leftarrow TriplePatternIterator(StartIterator(), p_p[tp_\epsilon], \phi_1^\epsilon)$
16: ▷ Za iterator vzorca trojčka vzamemo tisti vzorec trojčka, ki ima najmanj rezultatov.
17:
18: $I \leftarrow BasicGraphPatternIterator(I_\epsilon, \{p[tp] \mid tp \in B \setminus \{tp_\epsilon\}\}, \phi_1^\epsilon)$
19: **end while**
20: $p \leftarrow I.next()$
21: **end while**
22: **return** $p \cup p_p$
23: **end function**

Algoritem 2 Odjemalec - iterator vzorca trojčka

Podatki:

I_p = prednik ▷ Starševski iterator.
tp = vzorec trojčka
ϕ = trenutni delec (na začetku predstavlja prazno stran brez kontrol)
$I = nil$

```
1:  function TriplePatternIterator.next()
2:      while φ ne vsebuje neprebranih trojčkov do
3:          if φ ima povezavo na naslednjo stran u_φ then
4:              φ ← pridobi (HTTP GET) naslednjo stran preko u_φ
5:          else
6:              self.p_s ← I_p.next()
7:              if self.p_s = nil then
8:                  return nil
9:                    ▷ Konec iteratorja.
10:             end if
11:             φ ← pridobi (HTTP GET) prvo stran delca self.p_s[tp]
12:         end if
13:     end while
14:     t ← neprebran trojček iz φ
15:     p ← preslikava do rešitve, tako da je p[tp] = t
16:       ▷ Domena preslikave so spremenljivke trojčka tp.
17:     return p ∪ self.p_s
18: end function
```

3.1.4 Uporaba

Članek [42] ugotavlja, da so bili delci vzorcev trojčkov DBpedije uporabljeni 16,8 milijonkrat (število zahtevanih delcev) v prvih 9 mesecih, dosegljivost strežnika pa je bila 99,99 %, kar glede na članek [13] pomeni, da opazovana dostopna točka z delci vzorcev trojčkov po dosegljivosti spada v zgornjo tretjino najbolj zanesljivih oddaljenih dostopnih točk SPARQL. Največ dosto-

pov do strežnika je bilo (pričakovano) z odjemalcem delcev vzorcev trojčkov, ki tudi omogoča izvajanje poizvedb SPARQL.

Zanimivost je tudi to, da so bili najbolj pogosti naslednji vzorci trojčkov (po vrsti od najpogostejšega):

- `?s rdf:type <o>`: *subjekti* določenega tipa (`<o>` predstavlja določen objekt - URI ali literal),
- `?s rdf:type ?o`: *subjekti*, ki imajo tip (z zahtevkom tega delca se je merila dosegljivost),
- `?s ?p ?o`: delec z *vsemi* trojčki (začetna stran in začetna točka pri izvajanju poizvedb SPARQL),
- `<s> rdfs:subClassOf ?o`: podrazredi določenega *subjekta*.

Uporabljen je bil posredniški predpomnilniški strežnik nginx [2], čas veljavnosti posamezne strani pa je bil nastavljen na 1 uro. Izkazalo se je, da je 59,5 % zahtevanih delcev že bilo v predpomnilniku, torej pri več kot polovici zahtevkov strežnik delcev vzorcev ni bil obremenjen.

3.2 Uporaba relacijske baze za hrambo trojčkov

V sklopu našega dela smo implementirali preprost vmesnik, ki omogoča uporabo relacijske podatkovne baze MariaDB kot izvor podatkov, iz katerega črpa strežnik delcev vzorcev trojčkov. Trojčki so v našem primeru v relacijski bazi shranjeni v najbolj enostavni nenormalizirani obliki, tako da imamo tabelo s tremi stolpci `s`, `p` in `o`, ki predstavljajo *subjekt*, *predikat* in *objekt*. Vsak posamezen trojček je en zapis v tabeli, kot je prikazano v tabeli 3.1. Vsa polja so indeksirana. Način pridobivanja delca vzorca trojčka iz relacijske baze glede na poizvedbo, ki vsebuje *subjekt*, *predikat*, *objekt*, *zamik* (od katerega zaporednega zapisa oziroma trojčka naprej izbiramo podatke) in

omejitev (koliko trojčkov največ hočemo v rezultatih) je viden v algoritmu 3. Tehnični problemi, kot so pridobitev povezave in preoblikovanje zapisov, v psevdokodi niso zajeti. Funkcija `join` združi vse elemente (nize) v množici v en niz.

id	s	p	o
131665	record:10011	tags:taggedWithTag	tag:pop
131839	record:1003	tags:taggedWithTag	tag:pop
131842	record:1003	tags:taggedWithTag	tag:rock
131895	record:10031	tags:taggedWithTag	tag:rock
131964	record:10034	tags:taggedWithTag	tag:jazz
132091	record:1006	tags:taggedWithTag	tag:metal

Tabela 3.1: Izsek podatkov iz relacijske baze, v kateri hranimo zbirko Jamendo. Zaradi berljivosti so URI-ji v tabeli skrajšani z istimi predponami kot pri poizvedbi 3.2.

Algoritem 3 Funkcija, ki iz relacijske baze dobi delec vzorca trojčka.

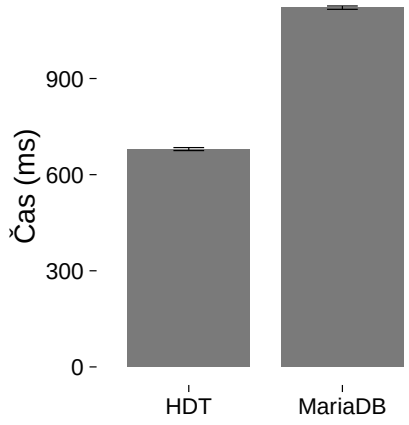
Podatki:

$db, tabela$ = podatkovna baza in tabela s trojčki

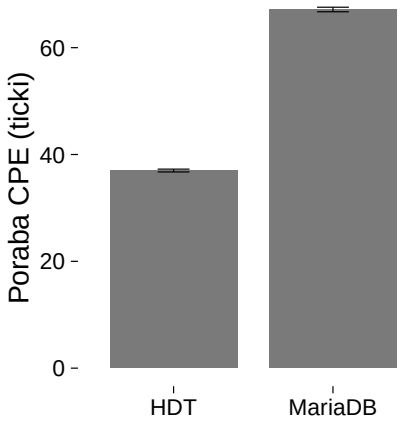
1: **function** IzvediPoizvedbo(poizvedba)
2: $pogoji \leftarrow \emptyset$
3: **if** $poizvedba.subjekt \neq nil \wedge poizvedba.subjekt[0] \neq$ "?" **then**
4: ▷ Izraz $spr[0]$ predstavlja prvi znak niza spr.
5: ▷ Če se del trojčka začne z "?", potem gre za spremenljivko.
6: $pogoji \leftarrow pogoji \cup (\text{“}s = \text{”} + poizvedba.subjekt)$
7: **else if** $poizvedba.predikat \neq nil \wedge poizvedba.predikat[0] \neq$ "?" **then**
8: $pogoji \leftarrow pogoji \cup (\text{“}p = \text{”} + poizvedba.predikat)$
9: **else if** $poizvedba.objekt \neq nil \wedge poizvedba.objekt[0] \neq$ "?" **then**
10: $pogoji \leftarrow pogoji \cup (\text{“}o = \text{”} + poizvedba.objekt)$
11: **end if**
12: $pogojiSql \leftarrow$ ""
13: **if** $|pogoji| > 0$ **then**
14: $pogojiSql \leftarrow$ " WHERE " $+ pogoji.join(\text{“ AND ”})$
15: **end if**
16: $poizvedbaSql \leftarrow$ "SELECT * FROM " $+ db +$ "." $+ tabela + pogojiSql$
17: **if** $poizvedba.zamik \neq nil \vee poizvedba.omejitev \neq nil$ **then**
18: ▷ Zamik določa, od katerega zapisa naprej hočemo rezultate.
19: ▷ Omejitev določa, koliko rezultatov največ hočemo.
20: $poizvedbaSql \leftarrow poizvedbaSql +$ " LIMIT " $+ (poizvedba.zamik \vee 0) +$ "," $+ (poizvedba.omejitev \vee 0)$ ▷ $spr \vee 0$ pomeni, da dobimo 0, če je $spr = nil$
21: **end if**
22: $countPoizvedbaSql \leftarrow$ "SELECT COUNT(*) AS stevilo FROM " $+ db +$ "." $+ tabela + pogojiSql$ ▷ Število vseh trojčkov (brez omejevanja).
23: $trojčki \leftarrow$ pridobi rezultate poizvedbe $poizvedbaSql$
24: $število \leftarrow$ pridobi rezultate poizvedbe $countPoizvedbaSql$
25: **return** trojčki (z omejitvijo) in število vseh trojčkov (brez omejitve) v pravi obliki s pomočjo podatkov v spremenljivkah $trojčki$ in $število$
26: **end function**

3.2.1 Performance pri uporabi relacijske baze MariaDB

Meritvam performanc pri izvajanju poizvedb SPARQL ob uporabi relacijske baze ne posvečamo veliko pozornosti, ker to ni osrednji del našega dela, ampak je precej obsežno področje, o katerem med drugim govorita članka [25] in [18]. V našem primeru gre le za preprosto filtriranje trojčkov v bazi, kot je opisano v tem poglavju, saj za izvajanje poizvedb SPARQL poskrbi odjemalec delcev vzorcev trojčkov, opisan v razdelku 3.1.3. Kot zanimivost smo izmerili čas izvajanja in porabo časa CPE pri izvajanju poizvedbe 3.2 nad zbirko Jamendo. Grafa na slikah 3.5 in 3.6 kažeta, da je za izvedbo poizvedbe 3.2 ob uporabi relacijske baze MariaDB potrebno precej več časa kot za izvedbo iste poizvedbe ob uporabi podatkovnega vira HDT.



Slika 3.5: Primerjava povprečnega časa izvajanja poizvedbe 3.2 nad zbirko Jamendo pri 20 izvajanjih z uporabo podatkovnih virov HDT in MariaDB.



Slika 3.6: Primerjava povprečne porabe časa CPE pri 20 izvajanjih poizvedbe 3.2 nad zbirko Jamendo z uporabo podatkovnih virov HDT in MariaDB.

Poglavje 4

Zmanjšanje števila vsebnostnih poizvedb s pomočjo metapodatkov

Zmanjšanje števila vsebnostnih poizvedb pri izvajanju poizvedb SPARQL preko delcev vzorcev trojčkov iz vidika ponudnika podatkov (strežnika) je možno na dva načina:

- **z razširitvijo izbirnika** – omogočimo bolj kompleksne poizvedbe oziroma filtre,
- **z dodajanjem metapodatkov** – dodamo več informacij o podatkih, ki jih lahko odjemalec uporabi za bolj učinkovito poizvedovanje.

V našem delu smo se osredotočili na slednji način; dodajanje metapodatkov. V primeru razširitve izbirnika bi namreč izgubili lastnost, opisano v razdelku 3.1.2, ki nam omogoča učinkovito izrabo posredniškega predpomnilniškega strežnika.

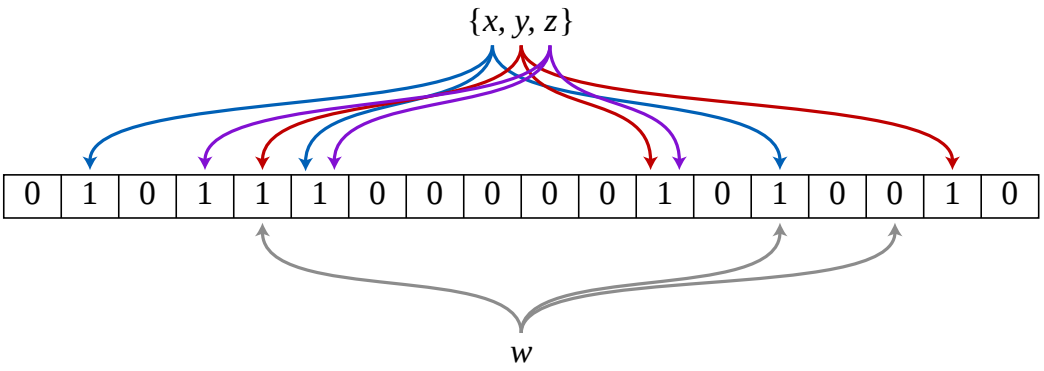
4.1 Bloomov filter

Bloomov filter [16] je prostorsko učinkovita verjetnostna podatkovna struktura, ki je uporabljena za preverjanje, če je poljuben element del množice. Lažni pozitivni odgovori so mogoči, lažni negativni odgovori pa ne. Na vprašanje, če je element v množici, lahko torej s pomočjo Bloomovega filtra odgovorimo z *"mogoče"* ali *"ne"*. Pri fiksni velikosti Bloomovega filtra se z večjim številom elementov v množici povečuje verjetnost lažnih pozitivnih odgovorov.

Prazen Bloomov filter je niz m bitov, kjer je vsak bit nastavljen na 0. Potrebujemo tudi k zgoščevalnih funkcij (angl. *hash functions*), od katerih vsaka preslika poljuben element na enega od m bitov. Oba parametra sta lahko odvisna od velikosti množice, za katero gradimo Bloomov filter. S podano fiksno želeno verjetnostjo lažnih pozitivnih odgovorov parameter m (velikost Bloomovega filtra) linearno narašča s številom elementov v množici [17].

Operacija *dodajanja* elementa je realizirana tako, da za dan element s pomočjo k zgoščevalnih funkcij pridobimo k mest v nizu in za ta mesta (ne glede na prejšnjo vrednost) nastavimo vrednost na 1.

Operacija *poizvedovanja*, če je element v množici, je realizirana tako, da (podobno kot pri dodajanju) za dan element s pomočjo k zgoščevalnih funkcij pridobimo k mest v nizu, nato pa, če so *vsa* pridobljena mesta nastavljena na 1, odgovorimo z *"mogoče"*, sicer pa z *"ne"*, saj bi bila vsa mesta nastavljena na 1, če bi takšen element prej dodali.



Slika 4.1: Bloomov filter. [3]

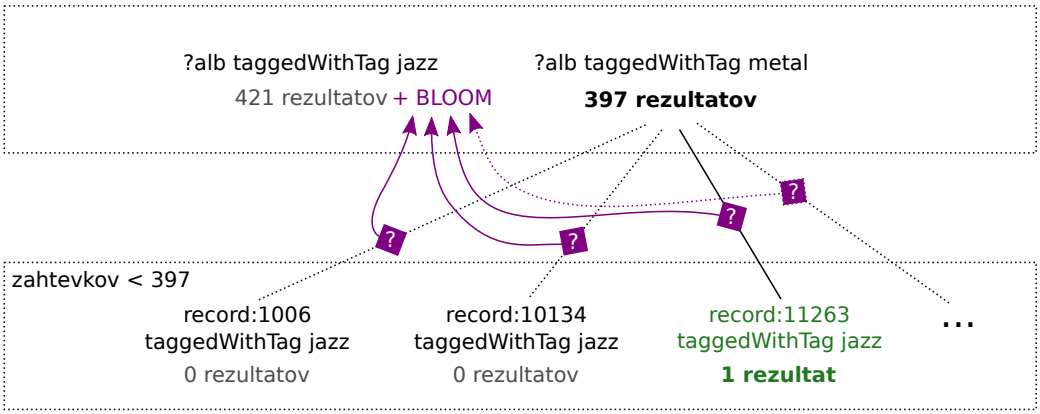
Slika 4.1 prikazuje Bloomov filter, ki vsebuje elemente $\{x, y, z\}$. Obarvane puščice prikazujejo mesta, kamor je vsak element preslikan. Kot je razvidno iz slike, imamo $k = 3$ (število zgoščevalnih funkcij) in $m = 18$ (velikost Bloomovega filtra). Spodaj je prikazan primer poizvedovanja po elementu w. Ker se w preslika na eno mesto, kjer je vrednost 0, vemo, da ga zagotovo ni v množici.

Bloomovi filtri se med drugim uporabljajo za podatkovne baze, usmerjevalne protokole in dostavljanje vsebine [29]. Zanimiv primer uporabe je tudi pri predpomnilniškem posredniškem strežniku, kjer se Bloomov filter uporablja zato, da strežnik pomni odgovore na zahtevke šele po drugem dostopu in se tako izogne pomnjenju odgovorov na zahtevke, ki se zgodijo le enkrat in nikoli več [19].

4.2 Razširitev AMF

Razširitev *AMF* (angl. *Approximate Membership Functions*), kot jo opisuje članek [40], na metapodatke delca vzorca trojčka doda verjetnostno podatkovno strukturo za množico, ki vsebuje *vse* trojčke, nastopajoče kot rezultati vzorca trojčka (ni razdelitve po straneh). Razširitev omogoča tudi uporabo podatkovne strukture GCS (angl. *Golomb-coded sets*) namesto Bloomovih filtrov, ki je prostorsko malo učinkovitejša, vendar porabi malo več procesorske

moči [33]. Medtem ko so rezultati razdeljeni po straneh, verjetnostna podatkovna struktura v metapodatkih obravnava celoten vzorec trojčka. Prednost te metode je, da zmanjša število vsebnostnih poizvedb (in posledično zahtevkov), saj nam ni potrebno preverjati vsebnosti tročkov, za katere verjetnostna podatkovna struktura v metapodatkih ne izkazuje (možnega) vsebovanja v celotnem delcu vzorca trojčka.



Slika 4.2: Prikaz izvedbe poizvedbe 3.1 z razširitvijo *AMF*. Namesto celotnih URI-jev so zaradi berljivosti prikazani zadnji deli le-teh.

Na sliki 4.2 je prikazana izvedba poizvedbe 3.1 preko delcev vzorcev trojčkov z opisano razširitvijo. Vijolične kvadratke si predstavljamo kot pogojne stavke, pri katerih vsebnostnih poizvedb ne naredimo, če, glede na Bloomov filter v metapodatkih, ni možnosti vsebovanja danega trojčka. Parametra Bloomovega filtra m (dolžina bitnega polja) in k (število zgoščevalnih funkcij) se nastavita glede na število rezultatov v celotnem delcu vzorca trojčka tako, da imamo fiksno verjetnost napake oziroma lažno pozitivnega odgovora. Privzeta vrednost je 0,001 oziroma 0,1 %, s katero, glede na meritve, dobimo dobre rezultate.

Vsak vzorec trojčka, ki je ob izvajanju poizvedbe nastal *po* vezavi spremenljivk, ima starševski delec vzorca trojčka (angl. *parent fragment*), ki predstavlja delec omenjenega vzorca trojčka pred vezavo spremenljivk. Različica, prenesena iz repozitorija, je imela problem pri pridobivanju starševskega

delca vzorca trojčka v primeru, ko sta več kot 2 vzorca trojčka v BGP-ju vsebovala isto spremenljivko. Zaradi načina izgradnje cevovoda iteratorjev so v slednjem primeru spremenljivke vezane, *preden* pride do koraka v iteratorju vzorca trojčka, kjer se preverja možnost vsebovanja trojčka v filtru. Ker imamo v tem koraku podatke o preslikavi spremenljivk, lahko enostavno naredimo inverz te preslikave. To smo naredili tako, da smo dodali funkcijo `removeBindings`, ki prejme vzorec trojčka in preslikave spremenljivk ter vrne prvoten vzorec, kakršen je bil pred preslikavo. Da torej dobimo pravi starševski delec, nad vzorcem trojčka izvedemo to funkcijo. Bolj elegantna rešitev bi sicer bila sprememba delovanja iteratorja vzorca trojčka, vendar je naša rešitev zaenkrat zadovoljiva. Po kontaktu z avtorji razširitve je bila naša rešitev dodana v repozitorij [38]. Povedali so, da sicer to ni bila napaka *per se*, vendar smo s tem posplošili njihovo rešitev in izboljšali delovanje. V meritvah in primerjavah je uporabljena že popravljena različica.

Članek govori tudi o oportunističnem poizvedovanju, kjer lahko na začetku prikažemo vse rezultate, za katere je, glede na verjetnostno podatkovno strukturo, *možno*, da so pravi in jih šele pozneje odstranimo, če ugotovimo, da je šlo za lažne pozitivne odgovore. Tako lahko dobimo delne rezultate *preden* naredimo potrebne vsebnostne poizvedbe za njih in tako izboljšamo uporabniško izkušnjo. Kot primer je podano iskanje po slikah, kjer lahko začnemo z vzporednim nalaganjem le-teh, preden naredimo potrebne vsebnostne poizvedbe. Te slike seveda lahko uporabniku skrijemo, dokler ne naredimo potrebnih vsebnostnih poizvedb.

Rezultat razširitve *AMF* je drastično zmanjšanje vsebnostnih poizvedb pri približno polovici od skupno 250 poizvedb, ustvarjenih s takratno različico testa WatDiv [12].

Poglavje 5

Opis naše razširitve za zmanjšanje števila vsebnostnih poizvedb

Tako kot razširitev *AMF* (razdelek 4.2) tudi naša razširitev uporablja verjetnostno podatkovno strukturo, bolj specifično Bloomove filtre, vendar pri naši razširitvi obravnavamo druge podatke. Naša razširitev na metapodatke delca vzorca trojčka doda Bloomov filter, ki vsebuje trojčke, povezane s podatki oziroma s *subjekti* in *objekti* trojčkov na *prikazani strani*. Katere trojčke pri tem dejansko vsebuje Bloomov filter, je odvisno od različice naše razširitve. Programska koda različice strežnika in odjemalca z našo razširitvijo, ki temelji na osnovni različici strežnika [39] in odjemalca [37] delcev vzorcev trojčkov, realizirani v jeziku JavaScript, je dostopna v repozitoriju GitHub [41].

5.1 Različice naše razširitve

Bloomov filter v metapodatkih pri naši razširitvi na strani i delca vzorca trojčka vsebuje:

- **če je spremenljivka vzorca trojčka *objekt*,** vse trojčke v zbirki, katerih objekt ali subjekt je *objekt* enega od trojčkov na strani i,
- **če je spremenljivka vzorca trojčka *subjekt*,** vse trojčke v zbirki, katerih objekt ali subjekt je *subjekt* enega od trojčkov na strani i,
- **če sta spremenljivki vzorca trojčka tako *subjekt* kot *objekt*,** imamo na voljo 4 različice:
    1. v metapodatke ne vključimo filtra (ta razširitev je označena kot *Jan 1*),
    2. vse trojčke v zbirki, katerih *objekt* ali *subjekt* nastopa kot *subjekt* vsaj enega od trojčkov na strani i (*Jan 2*),
    3. vse trojčke v zbirki, katerih *objekt* ali *subjekt* nastopa kot *objekt* vsaj enega od trojčkov na strani i (*Jan 3*),
    4. vse trojčke v zbirki, katerih *objekt* ali *subjekt* nastopa kot bodisi *subjekt* bodisi *objekt* vsaj enega od trojčkov na strani i (*Jan 4*),
- **v nasprotnem primeru (če je spremenljivka *predikat*)** v metapodatkih filter ni prisoten.

Dodatna možnost naše razširitve, ki zaradi berljivosti ni opisana v algoritmih, so različice, v tem delu imenovane *Jan N Filter*, pri čemer je N številka ene od zgoraj navedenih različic. Pri teh različicah v filter dodajamo le trojčke z določenimi *predikati* (torej filtriramo trojčke v verjetnostni strukturi glede na njihov predikat). S tem lahko dodatno zmanjšamo količino prenesenih podatkov, če vemo, kateri *predikati* se bodo pojavljali v vsebnostnih poizvedbah. Če npr. uporabniki večinoma izvajajo poizvedbe, kot je

poizvedba 3.2, ki proizvede vsebnostne poizvedbe s *predikatom* `tags:taggedWithTag`, potem za optimizacijo večine poizvedb v filtru potrebujemo samo trojčke z omenjenim predikatom, saj nam ostali le povečujejo velikost filtra in s tem prenos podatkov. Kateri *predikati* se pojavljajo v vsebnostnih poizvedbah, lahko ugotovimo s preprostim beleženjem in analizo zahtevkov. Po drugi strani imamo lahko na strani uporabnika višjenivojsko aplikacijo, za katero že vnaprej vemo tip poizvedb, ki jih bo le-ta izvajala in, posledično, kateri *predikati* se bodo pojavljali v vsebnostnih poizvedbah. Protokol za uskladitev seznama v filtru uporabljenih predikatov med strežnikom in odjemalcem ni določen; pri meritvah smo ta seznam ročno vnesli v programsko kodo strežnika in odjemalca, kot da gre za posebno različico, ki obravnava le določene predikate. Za splošno rabo bi bila potrebna sinhronizacija seznama predikatov pred vsakim izvajanjem poizvedbe, vendar bi bil gotovo del teh podatkov proti podatkom, potrebnih za izvedbo celotne poizvedbe, zelo majhen.

5.2 Podroben opis algoritmov

Za lažje razumevanje delovanja naše razširitve si poglejmo preprost primer. Podatki na spodnjih seznamih so predstavljivi s trojčki. Za boljšo berljivost si predstavljamo, da elementi množice {Album-A, Album-B, Album-C, Oseba-A, Oseba-B, Oseba-C, Oseba-D, nosi-oznako, je-na-voljo-na, je-izvajalec-na, CD, vinilka, metal, pop, rock, jazz} predstavljajo URI-je.

Imamo delec vzorca trojčka tp = `?alb nosi-oznako metal`, ki ima za *subjekt* spremenljivko. Trojčki, ki nastopajo v rezultatih, so:

- Album-A *nosi-oznako* metal,
- Album-B *nosi-oznako* metal,
- Album-C *nosi-oznako* metal.

V tem primeru bo Bloomov filter predstavljal množico z vsemi trojčki v

zbirki, v katerih kot *subjekt* ali *objekt* nastopajo *subjekti* zgornjih trojčkov, npr.:

- **Album-A *nosi-oznako* metal**,
- **Album-A *nosi-oznako* rock**,
- **Album-B *nosi-oznako* metal**,
- **Album-B *nosi-oznako* jazz**,
- **Album-C *nosi-oznako* pop**,
- **Album-C *nosi-oznako* metal**,
- Album-A *je-na-voljo-na* CD,
- Album-B *je-na-voljo-na* vinilka,
- Album-C *je-na-voljo-na* CD,
- Oseba-A *je-izvajalec-na* Album-A,
- Oseba-B *je-izvajalec-na* Album-B,
- Oseba-C *je-izvajalec-na* Album-B,
- Oseba-D *je-izvajalec-na* Album-C.

Če bi uporabili različico *Jan N Filter*, ki ima v seznamu predikatov le predikat `nosi-oznako`, potem bi v filtru nastopali le odebeljeni trojčki.

S psevdokodo je podrobno opisana različica *Jan 2*. Podroben postopek izgradnje filtra je opisan v algoritmu 5. Vrstica 2 je tista, ki je značilna za različico *Jan 2* in določa, da se v primeru, ko sta tako *objekt* kot *subjekt* spremenljivki, obravnava vse trojčke v zbirki, ki imajo *objekt* ali *subjekt* enak *subjektu* enega od trojčkov na strani — gledamo torej trojčke v zbirki, povezane s *subjekti* trojčkov na strani. Kot je razvidno, so parametri Bloomovega filtra (velikost bitnega polja in število preslikovalnih funkcij) v naši rešitvi nastavljeni v odvisnosti od velikosti množice, tako da imamo fiksno verjetnost

napake p_{napaka}, kar povzroči, da velikost filtra narašča linearno z velikostjo množice [17]. Ker je lahko velikost filtra prevelika, da bi se splačal prenos, imamo možnost, da ob preveliki količini trojčkov v filtru (fiksno določena zgornja omejitev), filtra ne zgradimo in ga posledično ne dodamo v metapodatke (glej vrstico 14).

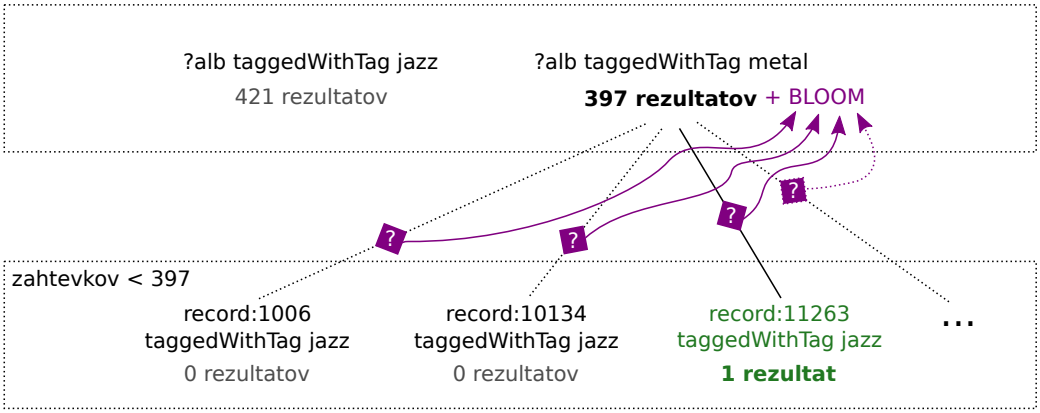
Naš odjemalec lahko s pomočjo Bloomovega filtra v metapodatkih naredi manj vsebnostnih poizvedb, saj mu ni potrebno preverjati obstoja trojčkov, povezanih z *objekti* oziroma *subjekti* rezultatov (odvisno od različice in oblike vzorca trojčka), za katere Bloomov filter ne izkazuje možnosti obstoja.

Čeprav parametra Bloomovega filtra m in k zasedeta sorazmerno malo prostora proti velikosti filtra, velja omeniti, da je način zapisa pri naši razširitvi zelo kompakten (niz znakov, ki ga vrača algoritem 5). Pri razširitvi *AMF* sta omenjena parametra z zapisom RDF podana posebej, ločeno od samega Bloomovega filtra, pri nas pa sta oba parametra združena v en niz, ki vsebuje tudi Bloomov filter (torej gre za samo en trojček). To nima velikega vpliva na količino prenesenih podatkov, zato bi bilo v nadaljnjem razvoju smiselno bolje definirati format zapisa pri naši razširitvi.

Na sliki 5.1 je prikazana izvedba poizvedbe 3.1 preko delcev vzorcev trojčkov z našo razširitvijo. Vijolične kvadratke si, na enak način kot pri razširitvi *AMF* predstavljamo kot pogojne stavke, pri katerih vsebnostnih poizvedb ne naredimo, če, glede na Bloomov filter v metapodatkih, ni možnosti vsebovanja danega trojčka. Pri omenjeni poizvedbi, kot je prikazano na sliki 5.1, preverjamo, če za vsak `?alb` v delcu vzorca trojčka tp_1 = `(?alb tags:taggedWithTag tag:metal)`, po vezavi *subjekta* na spremenljivko `?alb`, obstaja pripadajoč tp_2 = `(?alb tags:taggedWithTag tag:jazz)`. Bloomov filter se v naši rešitvi nanaša le na prikazano stran podatkov, zato bomo iz delca vzorca trojčka tp_1 v Bloomovih filtrih na *vseh* straneh zajeli podatke (efektivno trojčke), povezane s *subjekti* vseh trojčkov, ki nastopajo kot rezultati vzorca trojčka tp_1. Ker bomo zaradi planiranja poizvedbe (iterator BGP) zajeli le prvo stran delca vzorca trojčka tp_2, pri tem ne bomo prenesli vseh filtrov, vendar le filter, ki se nanaša na prvo stran. To ima

pozitiven učinek na zmanjšanje prenosa podatkov, saj so filtri v tp_2 pri naši rešitvi za izvajanje poizvedbe neuporabni. Podroben postopek preverjanja možnosti vsebnosti trojčkov pred izvedbo zahtevka z vsebnostno poizvedbo je opisan v algoritmu 4. Vrstica 13 je značilna za različico *Jan 2*, ker preverja, da *ni* spremenljivka v *objektu* tista, ki nastopa v vsebnostni poizvedbi, saj imamo pri tej različici v filtru le podatke, ki so povezani s *subjekti*. Kot je napisano v definiciji 1, tako v algoritmu 5 kot v algoritmu 4, $\mathcal{U}$ predstavlja množico URI-jev, $\mathcal{L}$ predstavlja množico literalov, $\mathcal{V}$ pa predstavlja množico spremenljivk.

Velikost Bloomovih filtrov, ki posledično vplivajo na količino prenesenih podatkov, je pri naši rešitvi za vsak BGP (množica vzorcev trojčkov B) odvisna *predvsem* od količine podatkov, povezanih s *subjekti* oziroma *objekti* trojčkov, ki nastopajo kot rezultati vzorca trojčka v množici B z najmanj rezultati. Velikost filtrov pri razširitvi *AMF* je, po drugi strani, odvisna od števila rezultatov *vseh* vzorcev trojčkov v B, vendar ne z njimi povezanimi podatki. Potrebna količina podatkov je pri vseh omenjenih razširitvah odvisna tako od podatkov kot od poizvedb, ki jih izvajamo nad njimi. Zaradi tega dejstva ne moremo preprosto odgovoriti na vprašanje, katera razširitev je boljša.



Slika 5.1: Prikaz izvedbe poizvedbe 3.1 z našo razširitvijo. Namesto celotnih URI-jev so zaradi berljivosti prikazane zadnje besede le-teh.

Algoritem 4 Odjemalec - Naša različica iteratorja vzorca trojčka

Podatki:

I_p = prednik ▷ Starševski iterator.
tp = vzorec trojčka
ϕ = trenutni delec ▷ Na začetku predstavlja prazno stran brez kontrol.
$\phi.filter$ = podatki Bloomovega filtra za delec ϕ
$\phi.tp$ = vzorec trojčka, kateremu pripada delec ϕ

1: **function** TriplePatternIterator.next()
2: $p \leftarrow nil$
3: **while** $p = nil$ **do**
4: **while** ϕ ne vsebuje neprebranih trojčkov **do**
5: **if** ϕ ima povezavo na naslednjo stran u_ϕ **then**
6: $\phi \leftarrow$ pridobi (HTTP GET) novo stran preko u_ϕ
7: **else**
8: $self.p_s \leftarrow I_p.next()$
9: **if** $self.p_s = nil$ **then**
10: **return** nil
11: **end if**
12: $tp_{vezan} \leftarrow self.p_s[tp]$
13: **if** $filter \neq nil \wedge tp_{vezan} \in \mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$
$\wedge((\phi.tp \in \mathcal{V} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L}) \vee \phi.tp \in \mathcal{U} \times \mathcal{U} \times \mathcal{V})$
$\vee(\phi.tp \in \mathcal{V} \times \mathcal{U} \times \mathcal{V} \wedge \phi.tp.objekt \notin \{tp.subjekt, tp.objekt\}))$
then
14: $filter \leftarrow filter.split(\text{“,”})$
15: $bloom \leftarrow$ nov bloom filter s parametroma $m = filter[0]$ in $k = filter[1]$ ter bitnim poljem $filter[2]$
16: $testStr \leftarrow tp_{vezan}.subjekt$ + “|”+ $tp_{vezan}.predikat$ + “|” + $tp_{vezan}.objekt$

Algoritem 4 Odjemalec - Naša različica iteratorja vzorca trojčkov - 2. del

```
17:                 if bloom.vsebuje(testStr) then
18:                     φ ← pridobi (HTTP GET) prvo stran delca tp_vezan
19:                     ▷ Ker so možni lažni pozitivni odgovori, moramo kljub
                          pozitivnem odgovoru narediti vsebnostno poizvedbo.
20:                 else
21:                     φ ← nil
22:                 end if
23:             else
24:                 φ ← pridobi (HTTP GET) prvo stran delca tp_vezan
25:             end if
26:         end if
27:     end while
28:     if φ ≠ nil then
29:         t ← neprebran trojček iz φ
30:         p ← preslikava do rešitve, tako da je p[tp] = t
31:             ▷ Domena preslikave so spremenljivke trojčka tp.
32:     else
33:         p ← nil
34:     end if
35: end while
36: return p ∪ self.p_s
37: end function
```

Algoritem 5 Strežnik - Naš algoritem za izgradnjo Bloomovega filtra

Podatki:

$p_{napaka} = 0{,}01$ ▷ Verjetnost napake.

$omejitev = nil$ ▷ Omejitev števila trojčkov v filtru, nil, če ni omejitve.

$tp =$ vzorec trojčka

$\phi =$ rezultati (trojčki) na trenutni strani

1: **function** VrniFilter()
2: **if** $tp \in \mathcal{V} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L}) \vee tp \in \mathcal{V} \times \mathcal{U} \times \mathcal{V}$ **then**
3: $spremenljivka \leftarrow subjekt$
4: **else if** $tp \in \mathcal{U} \times \mathcal{U} \times \mathcal{V}$ **then**
5: $spremenljivka \leftarrow objekt$
6: **else**
7: **return** nil
8: **end if**
9: **for all** $trojček \in \phi$ **do**
10: $q_1 \leftarrow$ poizvedba nad: $trojček[spremenljivka]$, `?p, ?o`
11: $q_2 \leftarrow$ poizvedba nad: `?s, ?p,` $trojček[spremenljivka]$
12: $rezultati \leftarrow rezultati \cup q_1.rezultati \cup q_2.rezultati$
13: **end for**
14: **if** $omejitev \neq nil \wedge |rezultati| > omejitev$ **then**
15: **return** nil
16: **end if**
17: $m \leftarrow \frac{\lceil -|rezultati| \cdot \ln p_{napaka} \rceil}{\ln 2 \cdot \ln 2}, k \leftarrow \left\lceil \frac{m}{|rezultati|} \cdot \ln 2 \right\rceil$
18: $bloom \leftarrow$ nov bloom filter s parametroma m in k
19: **for all** $trojček \in rezultati$ **do**
20: $bloom.dodaj(trojček.subjekt +$ "|"$+ trojček.predikat +$ "|"
$+ trojček.objekt)$
21: **end for**
22: $rezultat \leftarrow m +$ "," $+ k +$ "," $+ bloom.bitnopolje\_base64$
23: **return** $rezultat$
24: **end function**

Poglavje 6

Empirične meritve

6.1 Metodologija

Pri ocenjevanju uspešnosti razširitve smo obravnavali naslednje metrike:

- **Število zahtevkov do strežnika.** Z našo razširitvijo želimo zmanjšati število vsebnostnih poizvedb. Vsaka vsebnostna poizvedba predstavlja zahtevek HTTP, zato je število zahtevkov ena od glavnih metrik.
- **Količina prenesenih podatkov od strežnika proti odjemalcu.** Z zmanjšanjem števila vsebnostnih poizvedb se zmanjša količina prenesenih podatkov, vendar pa filter v metapodatkih poveča velikost delcev vzorcev trojčkov. Ker dandanes za dostop do informacij pogosto uporabjamo mobilne naprave, je ta metrika zelo pomembna. Pri naši razširitvi smo se osredotočili na optimizacijo iz vidika te metrike.
- **Poraba časa CPE na strežniku.** Ker izgradnja filtra terja procesorski čas, po drugi strani pa zmanjšanje vsebnostnih poizvedb pomeni manj zapravljenega procesorskega časa, smo pri vsaki poizvedbi izmerili količino porabljenega procesorskega časa na strani strežnika v *tickih* (angl. *ticks*).
- **Čas izvajanja poizvedbe.** Za vsako poizvedbo smo merili čas izvajanja v spletnem brskalniku Chromium.

Računalnik, na katerem je potekalo testiranje, ima CPE Intel Core i7-6700, ki teče s frekvenco 3,4 GHz, in 16 GiB delovnega pomnilnika. Testiranje je potekalo v operacijskem sistemu Ubuntu 16.04 LTS.

Pri vseh metrikah velja, da je manjša vrednost boljša, saj vsaka od njih predstavlja vir, ki ga hočemo pri izvajanju poizvedb čim manj izrabiti.

Odjemalec je napisan v jeziku JavaScript in deluje v večini, v času pisanja tega dela, priljubljenih modernih spletnih brskalnikih. Za avtomatizirano izvajanje meritev smo uporabili BrowserMob Proxy [1] in Selenium WebDriver [11] z brskalnikom Chromium.

Število zahtevkov in količino prenesenih podatkov smo izračunali s tolmačenjem datotek `HAR` [31] za vsako poizvedbo posebej.

Porabo časa CPE smo merili tako, da smo zajeli porabljen čas CPE za delovni proces (angl. *worker process*) strežnika delcev vzorcev trojčkov s preprostim tolmačenjem datoteke `/proc/PID/stat` [7], pri čemer je `PID` identifikator procesa (angl. *process identifier*). Število *tickov*, uporabljeno v naših meritvah, je izmerjeno tako, da je izračunana razlika seštevka $utime + ctime + cutime + cstime$ pred in po izvedbi posamezne poizvedbe. Pretvorbe nismo delali. V rezultatih so uporabljene številke, kot jih je vrnil sistem. To metriko obravnavamo zgolj kot približno oceno obremenitve strežniške CPE.

Čas je izmerjen v milisekundah od začetka do zaključka izvajanja posamezne poizvedbe. Pri tem strežnik teče na istem računalniku, zato so časi odgovorov na zahtevke manjši, kot bi bili v realnih okoliščinah, kjer bi število zahtevkov veliko bolj vplivalo na čas izvedbe.

Za testiranje smo uporabili Waterloo SPARQL Diversity Test Suite (WatDiv) [12] z zbirko, ki vsebuje 10 milijonov trojčkov. Različne razširitve in osnovno različico brez razširitev smo testirali s 340 poizvedbami SPARQL. Te poizvedbe so generirane z WatDiv tako, da je za vsako od predlog C1, F1, F2, F3, F4, F5, L1, L2, L3, L4, L5, S1, S2, S3, S4, S5 in S6[1] generiranih 20 poizvedb. Vsaka od teh predlog (predloge poizvedb 6.1, 6.2, 6.3, 6.4, 6.5,

[1]Predloge C2, C3 in S7 smo izpustili zaradi predolgega izvajanja. Meja časa izvajanja je bila okoli 300 sekund na poizvedbo.

6.6, 6.7, 6.8, 6.9, 6.10, 6.11, 6.12, 6.13, 6.14, 6.15, 6.16, 6.17) predstavlja določeno obliko poizvedb, naključno pa se spreminjajo določeni URI-ji. Enkrat npr. iščemo igralce, ki se nahajajo v mestu A, drugič pa igralce, ki se nahajajo v mestu B, sama oblika poizvedbe, če bi jo predstavili z grafom, pa ostaja enaka. V tabeli 6.1 je razvidno, kakšna je porazdelitev entitet (glede na tip) v zbirki. Izvedba poizvedb, ustvarjenih iz predlog F1, L2, L5, S2, S3, S4, S5 in S6, terja vsebnostne poizvedbe. Pri izvajanju teh poizvedb z našo razširitvijo pričakujemo manjše število zahtevkov, manjši prenos podatkov in (posledično) krajši čas izvajanja v primerjavi z osnovno različico. Pri ostalih poizvedbah pričakujemo malo večji prenos podatkov zaradi dodanega Bloomovega filtra. Verjetnost napake za Bloomove filtre je bila pri vseh razširitvah nastavljena na 0,1 %.

Na vseh stolpičnih grafih je prikazano povprečje obravnavane metrike pri izvedbi vseh poizvedb oziroma podmnožice le-teh. Pri intervalih zaupanja smo uporabili 5 % stopnjo tveganja. Zaradi boljšega približka porabe časa CPE in časa izvajanja poizvedb smo za vsako različico izvedli test 4-krat, torej smo izvedli $4 \cdot 340$ poizvedb za vsako različico. Količina prenesenih podatkov in število zahtevkov se nista spreminjala med ponavljanji testov iste različice, zato so pri grafih, ki predstavljajo ti dve metriki, uporabljeni podatki le ene izvedbe testa. Grafi pri porabi časa CPE in pri času izvajanja poizvedb predstavljajo povprečje vseh štirih izvedb testa. Temu primerno so tudi intervali zaupanja krajši.

Tam, kjer so uporabljene razširitve s predpomnilnikom za filtre, so meritve izvedene ob napolnjenem predpomnilniku, tako da so vsi filtri, ki nastopajo v odgovorih strežnika, vzeti iz predpomnilnika. Tam, kjer predpomnilnik ni uporabljen, so vsi filtri izračunani sproti.

Ker imamo odvisne vzorce (izvedba istih poizvedb SPARQL na istih podatkih z različnimi razširitvami), smo za potrditev statistično značilnih razlik med razširitvami uporabljali Wilcoxonov preizkus predznačenih rangov (angl. *Wilcoxon signed rank test*). Za stopnjo tveganja smo pri vseh testih uporabili $\alpha = 0{,}05$ (5 %). Povsod, kjer je $p < \alpha$, obravnavamo razliko kot statistično značilno.

```
SELECT ?v0 ?v4 ?v6 ?v7 WHERE {
  ?v0 sorg:caption ?v1 .
  ?v0 sorg:text ?v2 .
  ?v0 sorg:contentRating ?v3 .
  ?v0 rev:hasReview ?v4 .
  ?v4 rev:title ?v5 .
  ?v4 rev:reviewer ?v6 .
  ?v7 sorg:actor ?v6 .
  ?v7 sorg:language ?v8 .
}
```

Predloga poizvedbe 6.1: Predloga za poizvedbe C1. Vse poizvedbe, generirane s to predlogo, so enake.

```
SELECT ?v0 ?v2 ?v3 ?v4 ?v5 WHERE {
  ?v0 og:tag %v1% .
  ?v0 rdf:type ?v2 .
  ?v3 sorg:trailer ?v4 .
  ?v3 sorg:keywords ?v5 .
  ?v3 wsdbm:hasGenre ?v0 .
  ?v3 rdf:type wsdbm:ProductCategory2 .
}
```

Predloga poizvedbe 6.2: Predloga za poizvedbe F1. Pri tem se %v1% zamenja z naključno entiteto (URI) tipa `wsdbm:Topic` v zbirki.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 WHERE {
  ?v0 foaf:homepage ?v1 .
  ?v0 og:title ?v2 .
  ?v0 rdf:type ?v3 .
  ?v0 sorg:caption ?v4 .
  ?v0 sorg:description ?v5 .
  ?v1 sorg:url ?v6 .
  ?v1 wsdbm:hits ?v7 .
  ?v0 wsdbm:hasGenre %v8% .
}
```

Predloga poizvedbe 6.3: Predloga za poizvedbe F2. Pri tem se %v8% zamenja z naključno entiteto (URI) tipa `wsdbm:SubGenre` v zbirki.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 WHERE {
  ?v0 sorg:contentRating ?v1 .
  ?v0 sorg:contentSize ?v2 .
  ?v0 wsdbm:hasGenre %v3% .
  ?v4 wsdbm:makesPurchase ?v5 .
  ?v5 wsdbm:purchaseDate ?v6 .
  ?v5 wsdbm:purchaseFor ?v0 .
}
```

Predloga poizvedbe 6.4: Predloga za poizvedbe F3. Pri tem se %v3% zamenja z naključno entiteto (URI) tipa `wsdbm:SubGenre` v zbirki.

```
SELECT ?v0 ?v1 ?v2 ?v4 ?v5 ?v6 ?v7 ?v8 WHERE {
  ?v0 foaf:homepage ?v1 .
  ?v2 gr:includes ?v0 .
  ?v0 og:tag %v3% .
  ?v0 sorg:description ?v4 .
  ?v0 sorg:contentSize ?v8 .
  ?v1 sorg:url ?v5 .
  ?v1 wsdbm:hits ?v6 .
  ?v1 sorg:language wsdbm:Language0 .
  ?v7 wsdbm:likes ?v0 .
}
```

Predloga poizvedbe 6.5: Predloga za poizvedbe F4. Pri tem se %v3% zamenja z naključno entiteto (URI) tipa `wsdbm:Topic` v zbirki.

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 WHERE {
  ?v0 gr:includes ?v1 .
  %v2%  gr:offers ?v0 .
  ?v0 gr:price  ?v3 .
  ?v0 gr:validThrough ?v4 .
  ?v1 og:title  ?v5 .
  ?v1 rdf:type  ?v6 .
}
```

Predloga poizvedbe 6.6: Predloga za poizvedbe F5. Pri tem se %v2% zamenja z naključno entiteto (URI) tipa `wsdbm:Retailer` v zbirki.

```
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0 wsdbm:subscribes  %v1% .
  ?v2 sorg:caption  ?v3 .
  ?v0 wsdbm:likes ?v2 .
}
```

Predloga poizvedbe 6.7: Predloga za poizvedbe L1. Pri tem se %v1% zamenja z naključno entiteto (URI) `wsdbm:Website` v zbirki.

```
SELECT ?v1 ?v2 WHERE {
  %v0%  gn:parentCountry  ?v1 .
  ?v2 wsdbm:likes wsdbm:Product0 .
  ?v2 sorg:nationality  ?v1 .
}
```

Predloga poizvedbe 6.8: Predloga za poizvedbe L2. Pri tem se %v0% zamenja z naključno entiteto (URI) tipa `wsdbm:City` v zbirki.

```
SELECT ?v0 ?v1 WHERE {
  ?v0 wsdbm:likes ?v1 .
  ?v0 wsdbm:subscribes %v2% .
}
```

Predloga poizvedbe 6.9: Predloga za poizvedbe L3. Pri tem se %v2% zamenja z naključno entiteto (URI) tipa `wsdbm:Website` v zbirki.

```
SELECT ?v0 ?v2 WHERE {
  ?v0 og:tag %v1% .
  ?v0 sorg:caption ?v2 .
}
```

Predloga poizvedbe 6.10: Predloga za poizvedbe L4. Pri tem se %v1% zamenja z naključno entiteto (URI) tipa `wsdbm:Topic` v zbirki.

```
SELECT ?v0 ?v1 ?v3 WHERE {
  ?v0 sorg:jobTitle ?v1 .
  %v2% gn:parentCountry ?v3 .
  ?v0 sorg:nationality ?v3 .
}
```

Predloga poizvedbe 6.11: Predloga za poizvedbe L5. Pri tem se %v2% zamenja z naključno entiteto (URI) tipa `wsdbm:City` v zbirki.

```
SELECT ?v0 ?v1 ?v3 ?v4 ?v5 ?v6 ?v7 ?v8 ?v9 WHERE {
  ?v0 gr:includes ?v1 .
  %v2%  gr:offers ?v0 .
  ?v0 gr:price  ?v3 .
  ?v0 gr:serialNumber ?v4 .
  ?v0 gr:validFrom  ?v5 .
  ?v0 gr:validThrough ?v6 .
  ?v0 sorg:eligibleQuantity ?v7 .
  ?v0 sorg:eligibleRegion ?v8 .
  ?v0 sorg:priceValidUntil  ?v9 .
}
```

Predloga poizvedbe 6.12: Predloga za poizvedbe S1. Pri tem se %v2% zamenja z naključno entiteto (URI) tipa `wsdbm:Retailer` v zbirki.

```
SELECT ?v0 ?v1 ?v3 WHERE {
  ?v0 dc:Location ?v1 .
  ?v0 sorg:nationality  %v2% .
  ?v0 wsdbm:gender  ?v3 .
  ?v0 rdf:type  wsdbm:Role2 .
}
```

Predloga poizvedbe 6.13: Predloga za poizvedbe S2. Pri tem se %v2% zamenja z naključno entiteto (URI) tipa `wsdbm:Country` v zbirki.

```
SELECT ?v0 ?v2 ?v3 ?v4 WHERE {
  ?v0 rdf:type %v1% .
  ?v0 sorg:caption ?v2 .
  ?v0 wsdbm:hasGenre ?v3 .
  ?v0 sorg:publisher ?v4 .
}
```

Predloga poizvedbe 6.14: Predloga za poizvedbe S3. Pri tem se %v1% zamenja z naključno entiteto (URI) tipa `wsdbm:ProductCategory` v zbirki.

```
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0 foaf:age %v1% .
  ?v0 foaf:familyName ?v2 .
  ?v3 mo:artist ?v0 .
  ?v0 sorg:nationality wsdbm:Country1 .
}
```

Predloga poizvedbe 6.15: Predloga za poizvedbe S4. Pri tem se %v1% zamenja z naključno entiteto (URI) tipa `wsdbm:AgeGroup` v zbirki.

```
SELECT ?v0 ?v2 ?v3 WHERE {
  ?v0 rdf:type %v1% .
  ?v0 sorg:description ?v2 .
  ?v0 sorg:keywords ?v3 .
  ?v0 sorg:language wsdbm:Language0 .
}
```

Predloga poizvedbe 6.16: Predloga za poizvedbe S5. Pri tem se %v1% zamenja z naključno entiteto (URI) tipa `wsdbm:ProductCategory` v zbirki.

```
SELECT ?v0 ?v1 ?v2 WHERE {
  ?v0 mo:conductor  ?v1 .
  ?v0 rdf:type  ?v2 .
  ?v0 wsdbm:hasGenre  %v3% .
}
```

Predloga poizvedbe 6.17: Predloga za poizvedbe S6. Pri tem se %v3% zamenja z naključno entiteto (URI) tipa `wsdbm:SubGenre` v zbirki.

Tip entitete	Število primerkov
wsdbm:Purchase	1500
wsdbm:User	1000
wsdbm:Offer	900
wsdbm:Topic*	250
wsdbm:Product	250
wsdbm:City*	240
wsdbm:SubGenre*	145
wsdbm:Website	50
wsdbm:Language*	25
wsdbm:Country*	25
wsdbm:Genre*	21
wsdbm:ProductCategory*	15
wsdbm:Retailer	12
wsdbm:AgeGroup*	9
wsdbm:Role*	3
wsdbm:Gender*	2

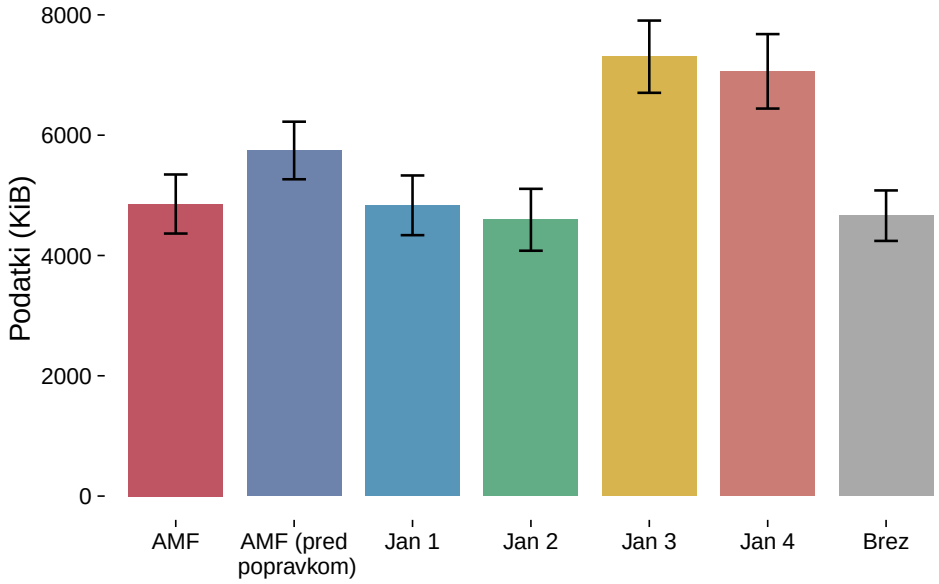
Tabela 6.1: Število primerkov entitet v zbirki pri faktorju 1. Naši testni podatki so ustvarjeni s faktorjem 100. Tipi entitet, označeni z zvezdico, se ne skalirajo, ostali pa se.

Pomen oznak razširitev *Jan 1*, *Jan 2*, *Jan 3* in *Jan 4* je opisan v poglavju 5.1. *AMF* predstavlja razširitev, opisano v poglavju 4.2, ki že vsebuje naš popravek oziroma izboljšavo, *AMF (pred popravkom)* pa predstavlja razširitev *AMF* brez naše izboljšave. Oznaka *Brez* predstavlja osnovno različico delcev vzorcev trojčkov, kot je opisana v razdelku 3.1.

6.2 Rezultati meritev

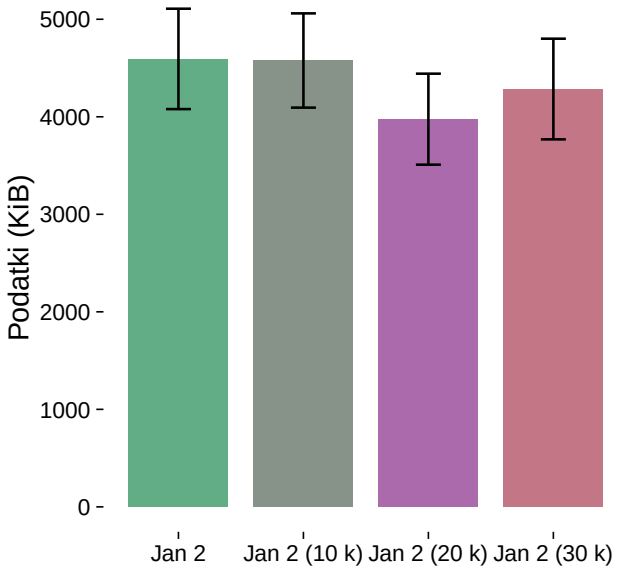
6.2.1 Količina prenesenih podatkov

Najbolj preprosta primerjava različic glede prenosa podatkov je primerjava povprečne količine prenesenih podatkov, ki je potrebna za izvedbo *vseh* 340 poizvedb SPARQL. Taka primerjava je z grafom prikazana na sliki 6.1. Od naših razširitev porabi najmanj podatkov *Jan 2*, zato bomo v nadaljevanju, tako iz vidika količine prenesenih podatkov kot iz vidika porabe časa CPE in časa izvajanja poizvedb, obravnavali to različico razširitve.



Slika 6.1: Primerjava povprečne količine prenesenih podatkov pri izvedbi *vseh* 340 poizvedb SPARQL z različnimi razširitvami.

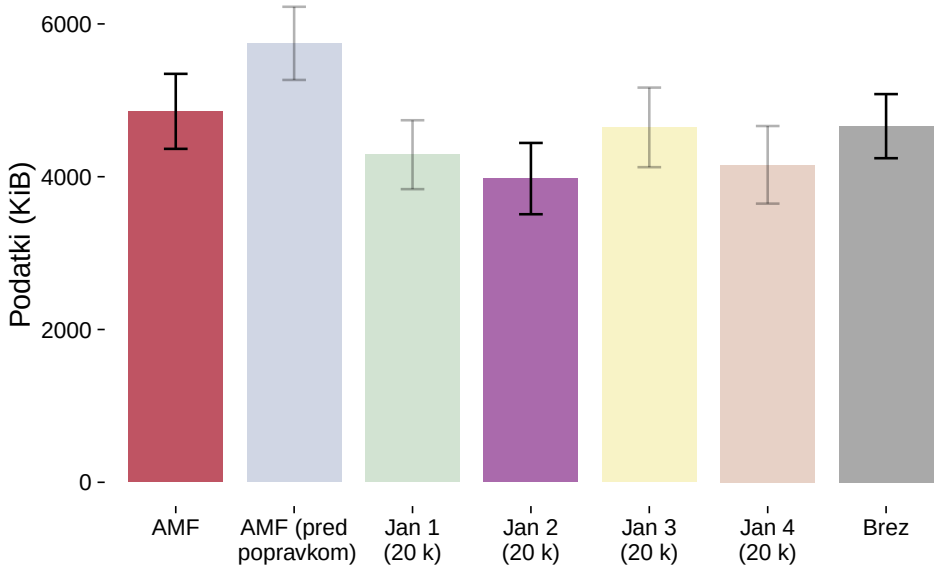
Količino prenesenih podatkov pri naši razširitvi smo poskušali zmanjšati s pomočjo uporabe zgornje omejitve števila trojčkov v filtru. Kot je razvidno iz vrstice 14 v algoritmu 5 ob prekoračitvi omenjene meje filtra ne pošiljamo. Ker je pri nas filter odvisen od trojčkov na eni strani, za poljuben vzorec trojčka prekoračitev na strani n ne implicira prekoračitve na strani $n + 1$. Graf, ki pokaže primerjavo količine prenesenih podatkov z razširitvijo *Jan 2* pri različnih omejitevah, je viden na sliki 6.2. Številka v oklepaju poleg oznake predstavlja omenjeno omejitev. Črka k je oznaka za *kilo* (10 k npr. predstavlja 10 tisoč). Ker, kot je razvidno iz grafa, *Jan 2* porabi najmanj podatkov, smo se odločili za splošno omejitev števila trojčkov v filtru pri naših razširitvah na 20 tisoč.



Slika 6.2: Primerjava povprečne količine prenesenih podatkov pri izvedbi *vseh* 340 poizvedb SPARQL z razširitvijo *Jan 2* in uporabo različnih omejitev števila trojčkov v filtru.

Na sliki 6.3 vidimo primerjavo povprečne količine prenesenih podatkov pri izvedbi *vseh* poizvedb, s tem da pri naših razširitvah uporabljamo omejitev števila trojčkov v filtru, nastavljeno na 20 tisoč. Razvidno je, da različica z

razširitvijo *Jan 2 (20 k)* prenese manj podatkov kot osnovna različica brez razširitev in manj kot različica z razširitvijo *AMF*. Statistično značilnost danih izjav poskusimo potrditi z Wilcoxonovima preizkusoma predznačenih rangov: $\overline{podatki_{Jan\,2\,(20\,k)}} = 3.975{,}417 < \overline{podatki_{Brez}} = 4.661{,}723$, pri čemer je $V = 23.764$ in $p = 0{,}998$; $\overline{podatki_{Jan\,2\,(20\,k)}} = 3.975{,}417 < \overline{podatki_{AMF}} = 4.855{,}591$, pri čemer je $V = 48.319$ in $p < 2{,}2 \cdot 10^{-16}$. Količina prenesenih podatkov pri uporabi razširitve *Jan 2 (20 k)* je statistično značilno manjša, kot pri razširitvi *AMF*, vendar ni statistično značilno manjša kot pri osnovni različici. Čeprav je povprečje pri metodi *Jan 2 (20 k)* precej manjše kot pri osnovni različici, pri malo več kot polovici sklopov (9 od 17) ni izboljšanja, ampak je blago poslabšanje. Gre za sklope, pri katerih ne delamo vsebnostnih poizvedb.



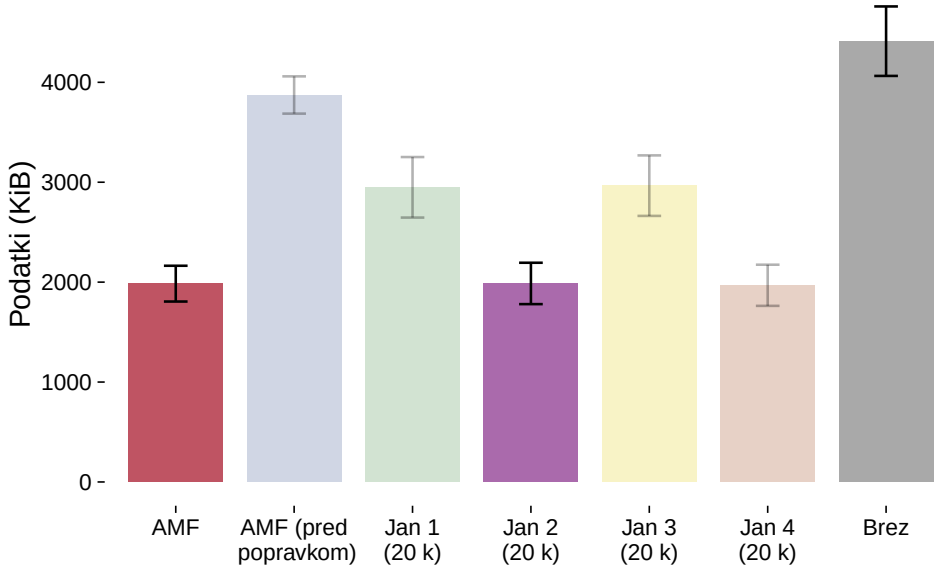
Slika 6.3: Primerjava povprečne količine prenesenih podatkov pri izvedbi *vseh* 340 poizvedb z različnimi razširitvami, s tem da pri naših razširitvah uporabljamo omejitev števila trojčkov v filtru.

Na sliki 6.4 vidimo primerjavo količine prenesenih podatkov pri izvedbi poizvedb, ki terjajo vsebnostne poizvedbe. Različica z razširitvijo *AMF* pri

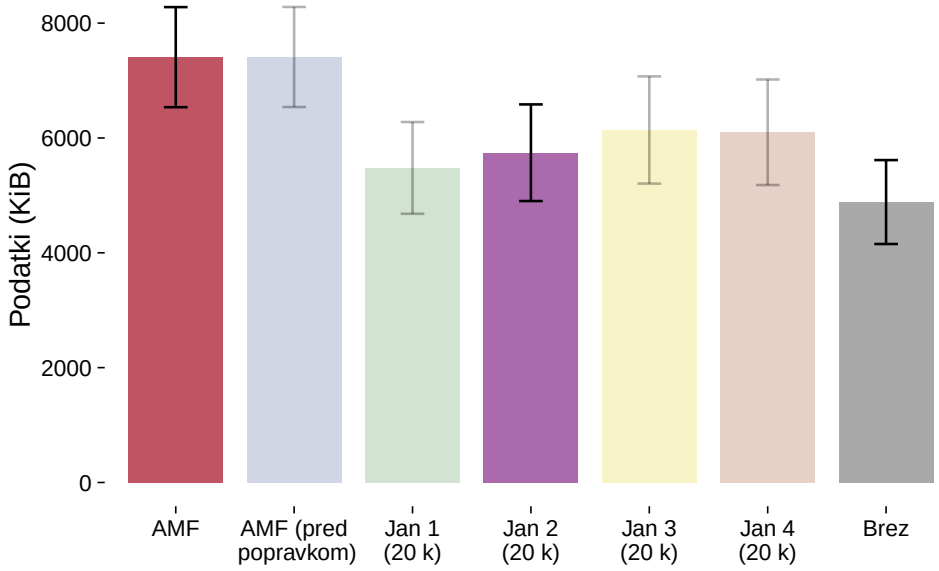
teh poizvedbah prenese malo manj podatkov kot različica z razširitvijo *Jan 2 (20 k)*, vendar razlika ni statistično značilna: $\overline{podatki_{AMF}} = 1.984{,}848 < \overline{podatki_{Jan\,2\,(20\,k)}} = 1.986{,}928$, pri čemer je $V = 6.709$ in $p = 0{,}6766$. Različici z razširitvama *AMF* in *Jan 2 (20 k)* pri izvajanju teh poizvedb preneseta veliko manj podatkov kot osnovna različica: $\overline{podatki_{AMF}} = 1.984{,}848 < \overline{podatki_{Brez}} = 4.411{,}614$, pri čemer je $V = 2.713$ in $p = 1{,}078 \cdot 10^{-10}$; $\overline{podatki_{Jan\,2\,(20\,k)}} = 1.986{,}928 < \overline{podatki_{Brez}} = 4.411{,}614$, pri čemer je $V = 2.902$ in $p = 8{,}305 \cdot 10^{-10}$.

Na sliki 6.5 vidimo primerjavo količine prenesenih podatkov pri izvedbi poizvedb, ki ne terjajo vsebnostnih poizvedb. Razvidno je, da različica z razširitvijo *Jan 2 (20 k)* ob izvedbi teh poizvedb prenese manj podatkov kot različica z razširitvijo *AMF*, vendar, kot pričakovano, več kot osnovna različica: $\overline{podatki_{Jan\,2\,(20\,k)}} = 5.742{,}963 < \overline{podatki_{AMF}} = 7.407{,}362$, pri čemer je $V = 1$ in $p < 2{,}2 \cdot 10^{-16}$; $\overline{podatki_{Brez}} = 4.884{,}043 < \overline{podatki_{Jan\,2\,(20\,k)}} = 5.742{,}963$, pri čemer je $V = 0$ in $p < 2{,}2 \cdot 10^{-16}$. Obe razširitvi dodata filter v metapodatke in s tem povečata velikost delcev vzorcev trojčkov. Ob odsotnosti vsebnostnih poizvedb se število zahtevkov pri nobeni omenjeni razširitvi ne zmanjša. Razširitev *Jan 2 (20 k)* v tem primeru prenese manj podatkov kot razširitev *AMF*, ker je pri nas filter razdeljen po straneh, v celoti (vse strani) pa je prenesen le tisti delec vzorca trojčka, ki ima najmanj rezultatov (glej visokonivojski algoritem v razdelku 3.1.3).

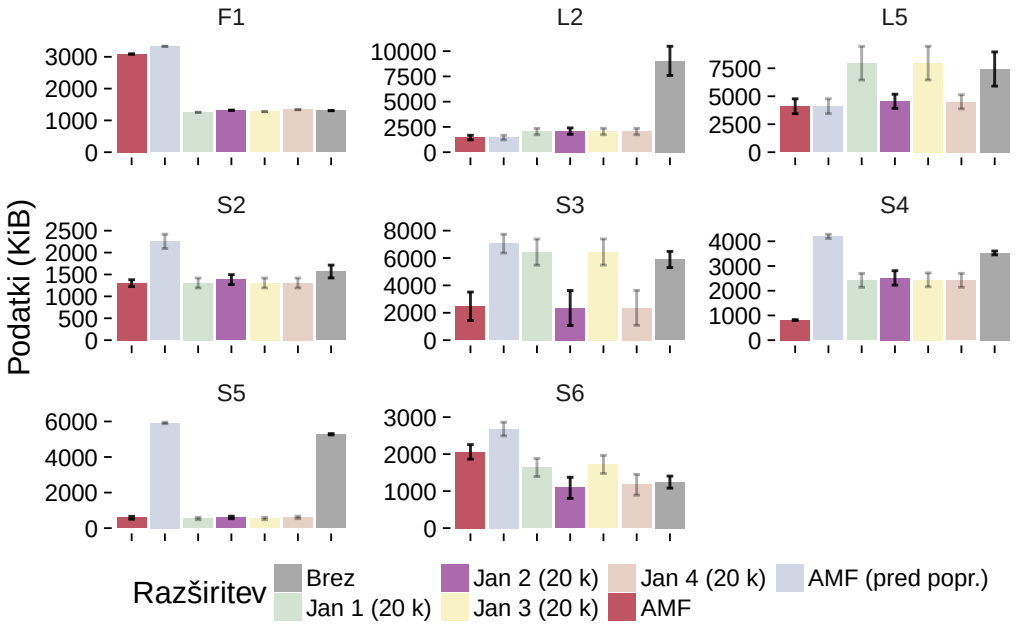
Na slikah 6.6 in 6.7 lahko vidimo po sklopih razdeljene primerjave povprečne količine prenesenih podatkov na poizvedbo. Na sliki 6.6 opazimo, da z uporabo razširitve *Jan 2 (20 k)* pri izvajanju poizvedb v sklopu S4 prenesemo znatno več podatkov kot z uporabo razširitve *AMF*.



Slika 6.4: Primerjava povprečne količine prenesenih podatkov pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), z različnimi razširitvami.



Slika 6.5: Primerjava povprečne količine prenesenih podatkov pri izvedbi sklopov poizvedb, ki *ne* terjajo vsebnostnih poizvedb (C1, F2, F3, F4, F5, L1, L3, L4, S1), z različnimi razširitvami.



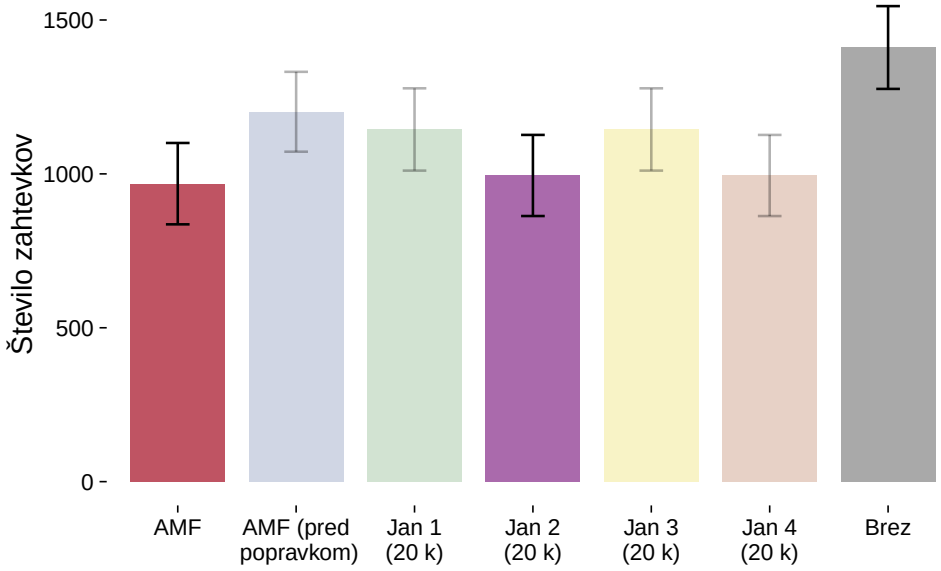
Slika 6.6: Primerjava povprečne količine prenesenih podatkov po sklopih, pri katerih delamo vsebnostne poizvedbe.

Slika 6.7: Primerjava povprečne količine prenesenih podatkov po sklopih, pri katerih *ne* delamo vsebnostnih poizvedb.

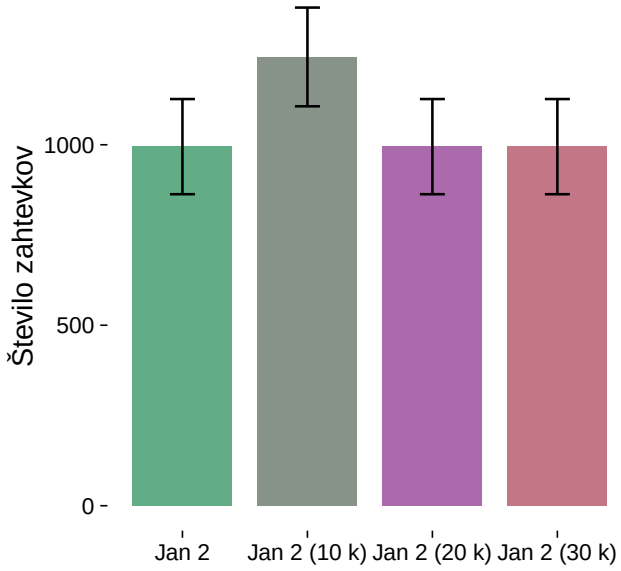
6.2.2 Število zahtevkov

Pri številu zahtevkov si, prav tako kot pri količini prenesenih podatkov, najprej ogledamo povprečne vrednosti po različnih pristopih. Gre za primerjavo povprečnega števila zahtevkov na poizvedbo pri izvedbi *vseh* 340 poizvedb, ki je ponazorjena z grafom na sliki 6.8. V tem primeru obravnavamo vse naše razširitve z omejitvijo 20 tisoč trojčkov v filtru.

Kot je razvidno iz grafa na sliki 6.9, se število zahtevkov ob uporabi razširitve *Jan 2* pri omejitvi 20 tisoč ali več ne poveča. Na razširitev *Jan 2 (20 k)* smo se osredotočili, ker želimo optimirati količino prenesenih podatkov.



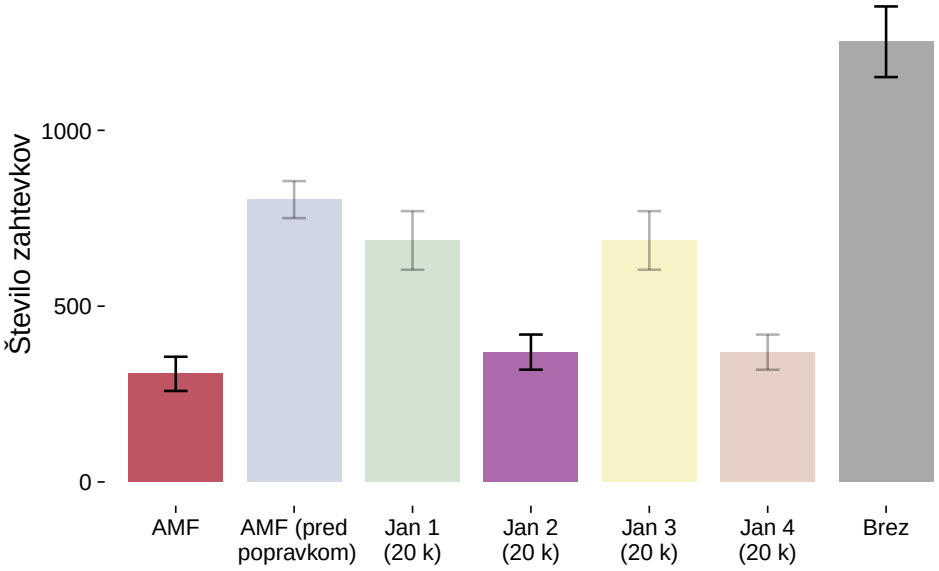
Slika 6.8: Primerjava povprečnega števila zahtevkov pri izvedbi *vseh* 340 poizvedb z različnimi razširitvami.



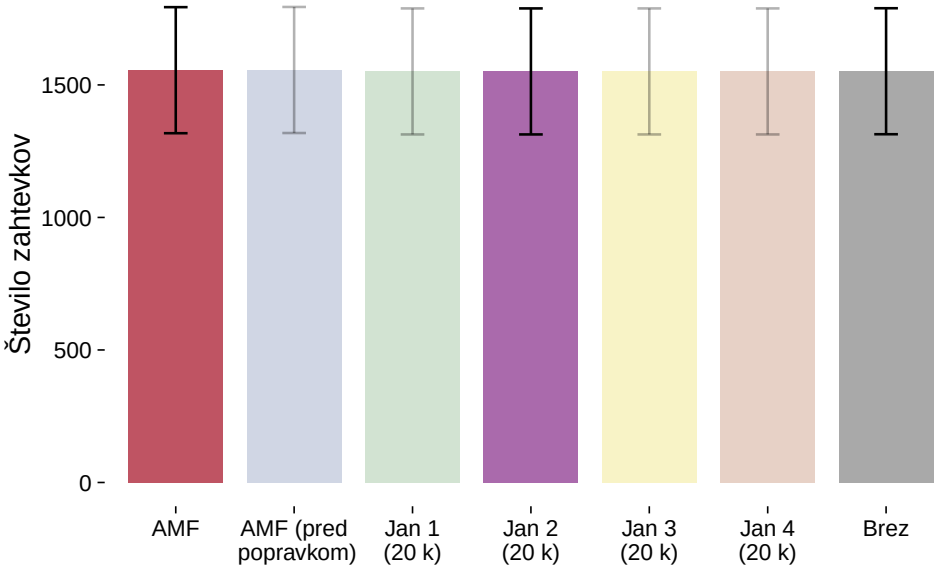
Slika 6.9: Primerjava povprečnega števila zahtevkov pri izvedbi *vseh* 340 poizvedb z razširitvijo *Jan 2* in uporabo različnih omejitev števila trojčkov v filtru.

Na sliki 6.10 vidimo primerjavo števila zahtevkov pri izvedbi poizvedb, ki terjajo vsebnostne poizvedbe. Različici z razširitvama *Jan 2 (20 k)* in *AMF* ob izvedbi teh poizvedb naredita manj zahtekov kot osnovna različica: $\overline{zahtevki_{Jan\,2\,(20\,k)}} = 369{,}2688 < \overline{zahtevki_{Brez}} = 1.251{,}95$, pri čemer je $V = 0$ in $p < 2{,}2 \cdot 10^{-16}$; $\overline{zahtevki_{AMF}} = 307{,}5 < \overline{zahtevki_{Brez}} = 1.251{,}95$, pri čemer je $V = 435$ in $p < 2{,}2 \cdot 10^{-16}$.

Na sliki 6.11 lahko opazimo, da se ob uporabi razširitev pri poizvedbah, ki ne terjajo vsebnostnih poizvedb, število zahtevkov ne zmanjša. To je seveda pričakovano, saj vse obravnavane razširitve poskušajo zmanjšati le število vsebnostnih poizvedb.

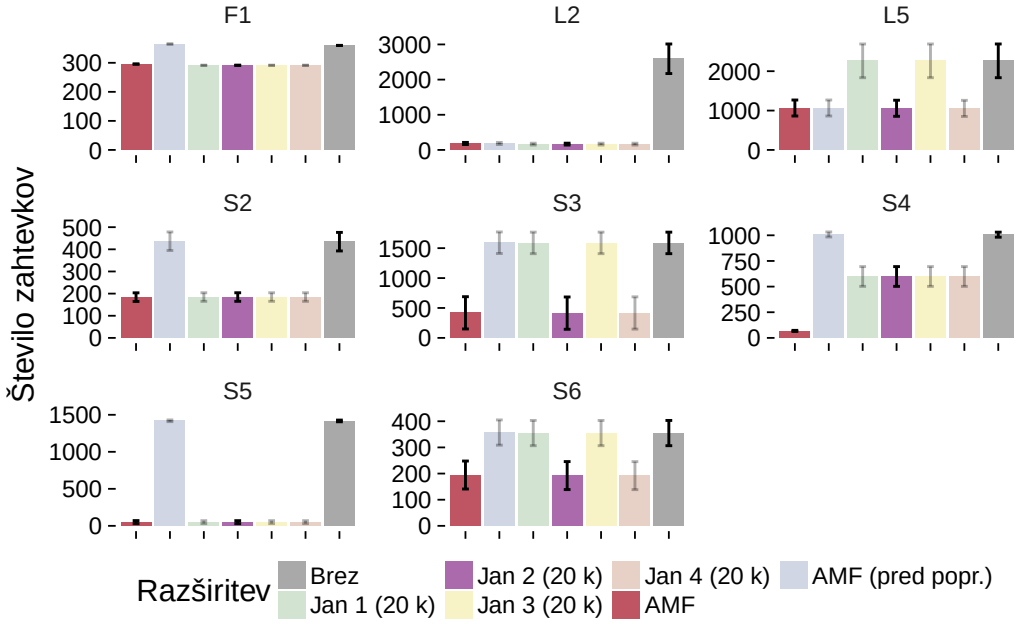


Slika 6.10: Primerjava povprečnega števila zahtevkov pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), z različnimi razširitvami.



Slika 6.11: Primerjava povprečnega števila zahtevkov pri izvedbi sklopov poizvedb, ki *ne* terjajo vsebnostnih poizvedb (C1, F2, F3, F4, F5, L1, L3, L4, S1), z različnimi razširitvami.

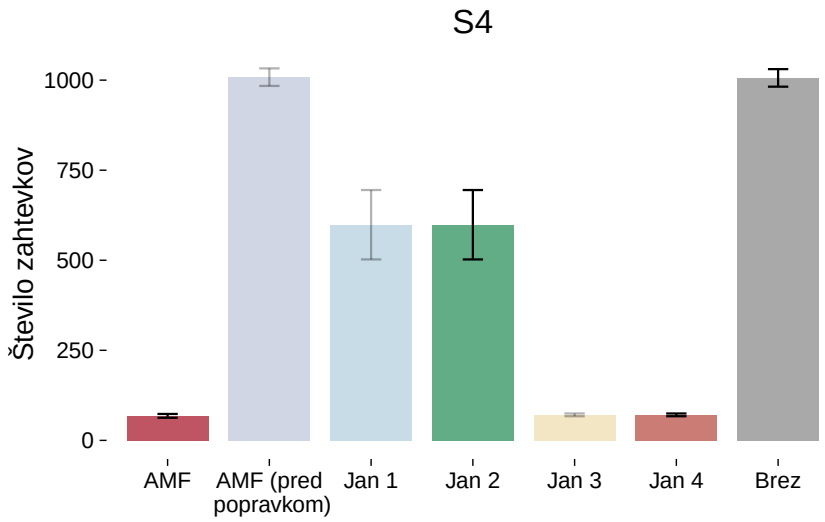
Na sliki 6.12 lahko vidimo po sklopih razdeljene primerjave povprečnega števila zahtevkov na poizvedbo.



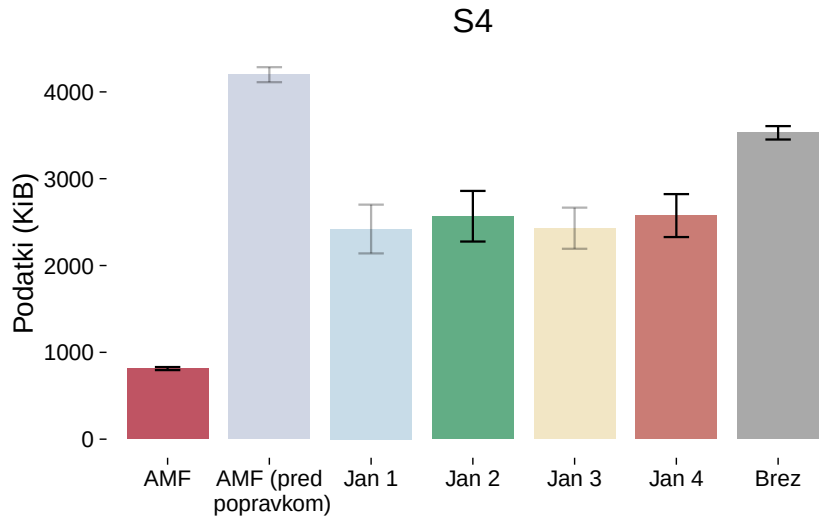
Slika 6.12: Primerjava povprečnega števila zahtevkov po sklopih, pri katerih delamo vsebnostne poizvedbe.

Kot vidimo na sliki 6.12, se pri izvajanju poizvedb sklopa S4, glede na število zahtevkov, razširitev *AMF* obnese statistično značilno bolje kot razširitev *Jan 2 (20 k)* (in tudi, kot je razvidno iz slike, ostale naše razširitve z omejitvijo 20 k): $\overline{zahtevki_{AMF}} = 68 < \overline{zahtevki_{Jan\,2\,(20\,k)}} = 598{,}6$, pri čemer je $V = 36$ in $p = 0{,}004796$. Kakor lahko opazimo na grafu na sliki 6.13, se ob odstranitvi omejitve trojčkov v filtru število zahtevkov pri razširitvah *Jan 3* in *Jan 4* drastično zmanjša. Zaradi oblike poizvedb in načina njihovega izvajanja v sklopu S4 sta razširitvi *Jan 3* in *Jan 4* glede števila zahtevkov boljši kot razširitev *Jan 2*, vendar pa pri količini prenesenih podatkov, kot kaže graf na sliki 6.14, zaradi velikosti filtrov, ni znatne izboljšave. Tudi v primeru odsotnosti omejitve trojčkov v filtru, različica z razširitvijo *Jan 4* pri izvajanju

poizvedb iz sklopa S4 prenese statistično značilno več podatkov kot različica z razširitvijo *AMF*: $\overline{podatki_{AMF}} = 813{,}0699 < \overline{podatki_{Jan\,4}} = 2.575{,}228$, pri čemer je $V = 0$ in $p = 4{,}225 \cdot 10^{-5}$. Kot je razvidno iz grafa na sliki 6.12, se pri razširitvah *Jan 3 (20 k)* in *Jan 4 (20 k)* zaradi omejitve velikosti filtra dogaja, da filtri niso poslani in imamo posledično, kar se tiče števila zahtevkov, slabše rezultate. Za ta primer so torej naše razširitve slabše kot razširitev *AMF*. Ob predpostavki, da vnaprej vemo, kakšne poizvedbe se bodo izvajale, lahko uporabljamo eno od razširitev *Jan N Filter*, npr. *Jan 4 Filter*. Rezultati meritev pri uporabi slednje razširitve so predstavljeni v razdelku 6.3.



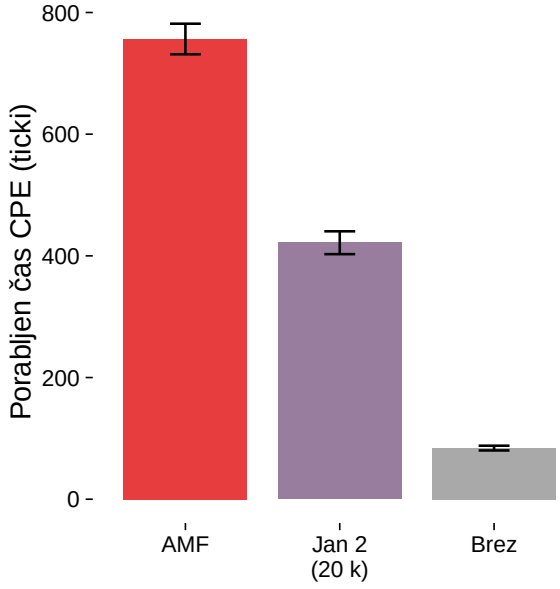
Slika 6.13: Primerjava povprečnega števila zahtevkov pri izvedbi poizvedb sklopa S4.



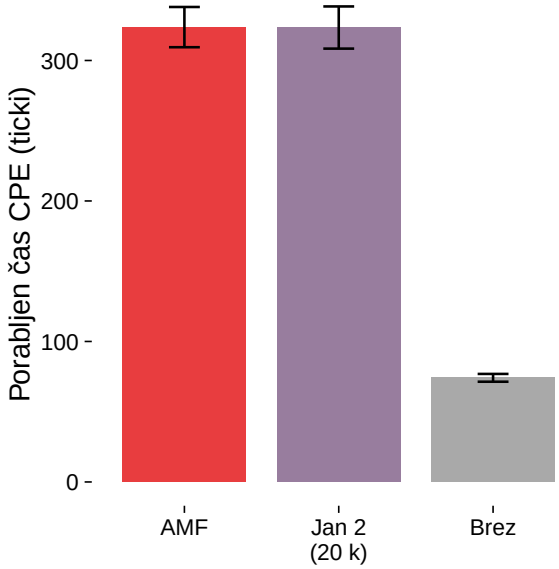
Slika 6.14: Primerjava povprečne količine prenesenih podatkov pri izvedbi poizvedb sklopa S4.

6.2.3 Poraba časa CPE na strežniku

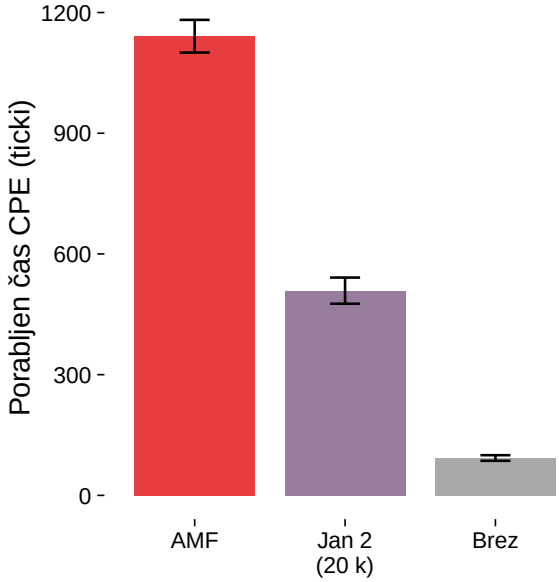
Kot vidimo na sliki 6.15, je povprečna poraba časa CPE pri izvedbi *vseh* 340 poizvedb najmanjša, če ne uporabljamo nobenih razširitev. Različica z razširitvijo *Jan 2 (20 k)* pri izvajanju *vseh* poizvedb porabi manj časa CPE kot različica z razširitvijo *AMF*. Razlika je statistično značilna: $\overline{cpe_{Jan\,2\,(20\,k)}} = 421{,}5787 < \overline{cpe_{AMF}} = 756{,}4493$, pri čemer je $V = 277.660$ in $p < 2{,}2 \cdot 10^{-16}$. Iz grafov na slikah 6.16 in 6.17 je razvidno, da je razlika med razširitvama *Jan 2 (20 k)* in *AMF* prisotna pri izvedbi poizvedb, ki *ne* terjajo vsebnostnih poizvedb, medtem ko pri poizvedbah, ki terjajo vsebnostne poizvedbe, ni velike razlike.



Slika 6.15: Primerjava povprečnega časa CPE pri izvedbi *vseh* 340 poizvedb.



Slika 6.16: Primerjava povprečnega časa CPE pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6).

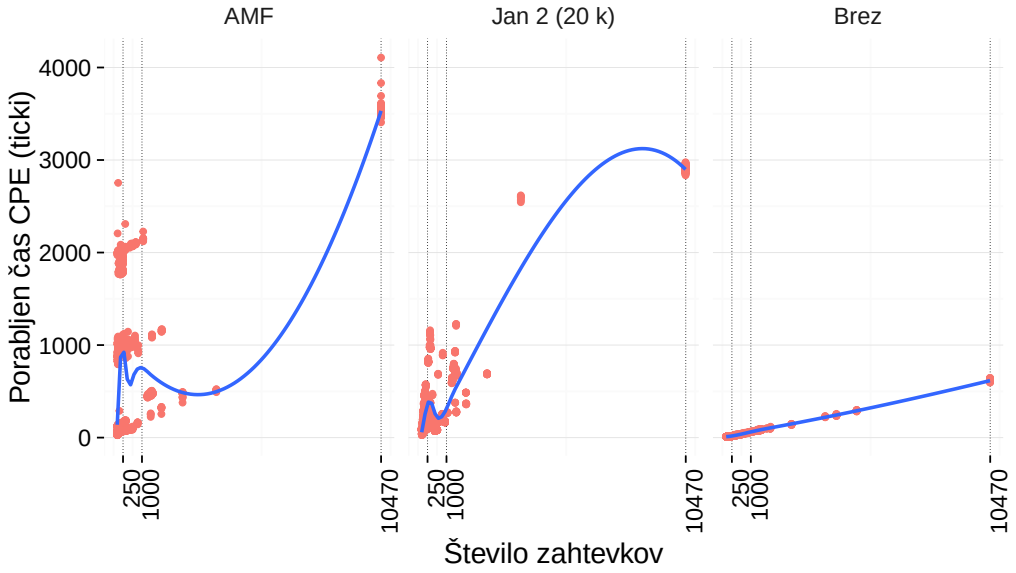


Slika 6.17: Primerjava povprečnega časa CPE pri izvedbi sklopov poizvedb, ki *ne* terjajo vsebnostnih poizvedb (C1, F2, F3, F4, F5, L1, L3, L4, S1).

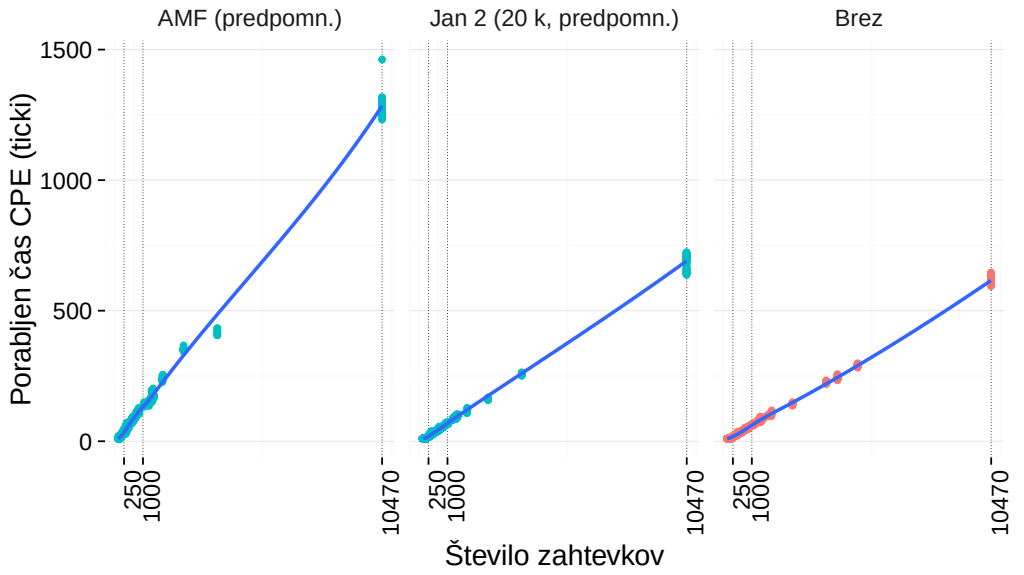
Na sliki 6.18 vidimo, kako se za izvedbo posamezne poizvedbe spreminja

poraba CPE glede na število zahtevkov, ki jih zaradi njenega izvajanja naredimo. Povprečje števila zahtevkov pri *AMF* in *Jan 2 (20 k)* je okoli 1000, mediana pa je okoli 250. Ti dve vrednosti sta pri obeh grafih na slikah 6.18 in 6.19 označeni na osi x. Krivulji na obeh grafih sta zglajeni z metodo LOESS [23].

Poraba CPE časa se približa osnovni različici brez razširitev, če uporabljamo predpomnilnik *in so vsi filtri že v predpomnilniku*. Slika 6.19 prikazuje porabo časa CPE v odvisnosti od števila zahtevkov ob uporabi predpomnilnika pri razširitvah *AMF* in *Jan 2 (20 k)* ter pri osnovni različici. Razlika med razširitvama *AMF* in *Jan 2 (20 k)* ob uporabi predpomnilnika nastane predvsem zaradi različnih načinov delovanja predpomnjenja. Če bi uporabljali isti način predpomnjenja, bi bila razlika v porabi časa CPE manjša.



Slika 6.18: Poraba povprečnega časa CPE glede na število zahtevkov, potrebnih za izvedbo poizvedbe. Skala je linearna.

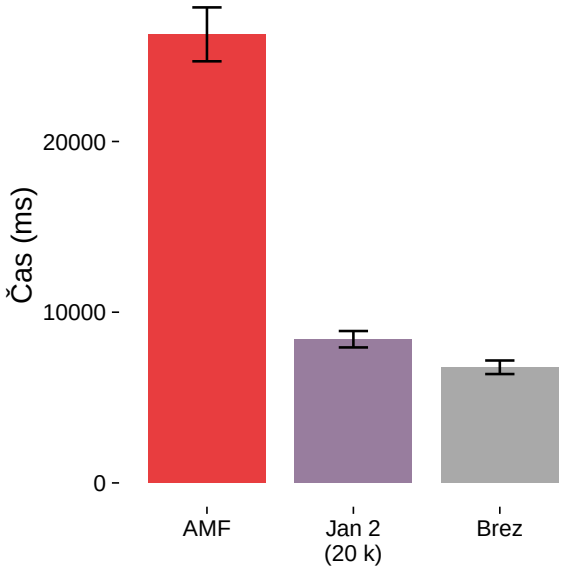


Slika 6.19: Poraba povprečnega časa CPE glede na število zahtevkov, potrebnih za izvedbo poizvedbe, s tem da razširitvi *AMF* in *Jan 2* uporabljata predpomnilnik za filtre. Skala je linearna.

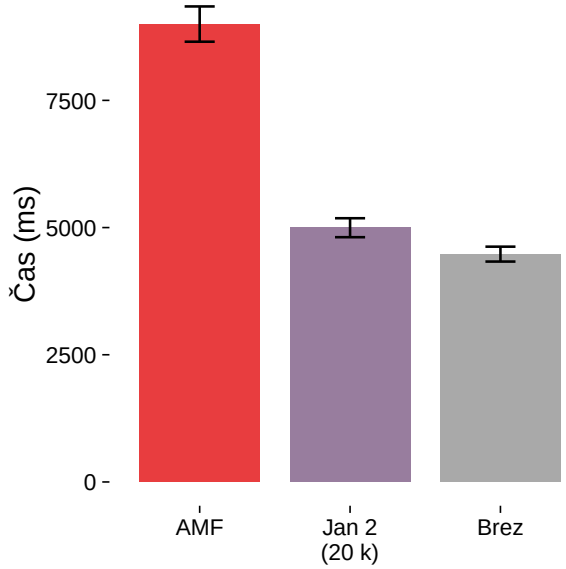
6.2.4 Čas izvajanja poizvedb

Če primerjamo povprečen čas izvajanja *vseh* 340 poizvedb (graf na sliki 6.20), ugotovimo, da je kljub manjšemu številu zahtevkov in manjši količini prenesenih podatkov najhitrejša osnovna verzija brez razširitev. Na tem mestu se moramo zavedati, da je testiranje potekalo lokalno, tako da je bil strežnik na istem računalniku kot brskalnik z odjemalcem. Pri realnem scenariju bi bil čas izvedbe precej bolj odvisen od števila zahtevkov in količine prenesenih podatkov, saj bi v tem primeru do izraza prišla hitrost povezave med strežnikom in odjemalcem. Razširitev *Jan 2 (20 k)* nudi hitrejše poizvedovanje kot razširitev *AMF*: $\overline{čas_{Jan\,2\,(20\,k)}} = 8.416{,}918 < \overline{čas_{AMF}} = 26.274{,}82$, pri čemer je $V = 73.844$ in $p < 2{,}2 \cdot 10^{-16}$. En razlog za razliko v hitrosti je, da imamo pri različici z razširitvijo *Jan 2 (20 k)* namesto večjih filtrov več manjših filtrov, razdeljenih po straneh in posledično pri preverjanju možnega

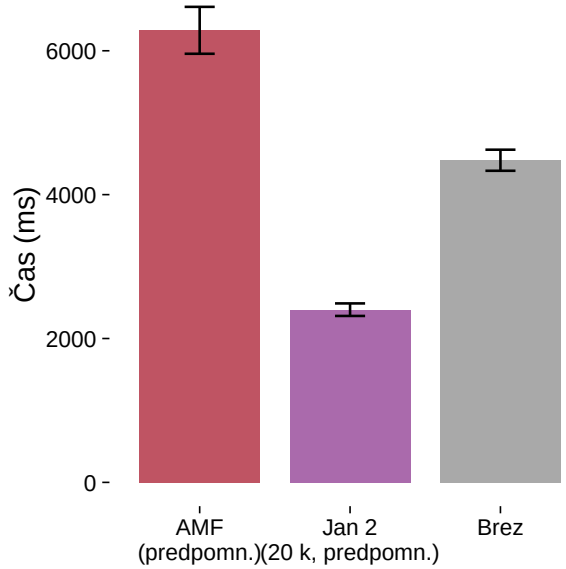
vsebovanja za vsak trojček računamo z manjšim številom zgoščevalnih funkcij, medtem ko pri razširitvi *AMF* delamo z daljšimi filtri, ki vsebujejo trojčke celotnih delcev.



Slika 6.20: Primerjava povprečnega časa izvajanja pri izvedbi *vseh* 340 poizvedb.

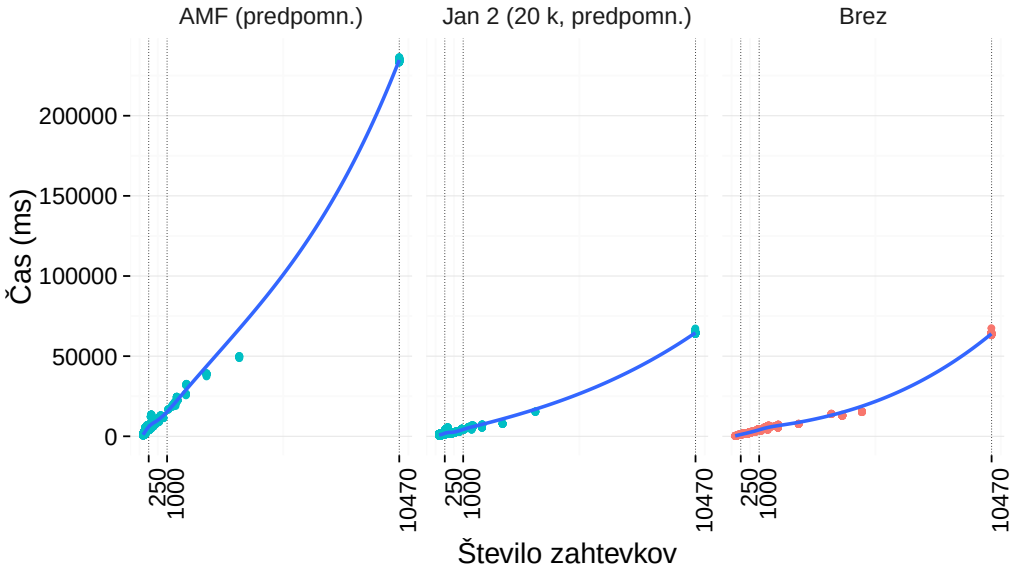


Slika 6.21: Primerjava povprečnega časa izvajanja pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6).



Slika 6.22: Primerjava povprečnega časa izvajanja pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), s tem da razširitvi *AMF* in *Jan 2* uporabljata predpomnilnik za filtre.

Na sliki 6.23 vidimo povprečno porabo časa na poizvedbo v odvisnosti od števila zahtevkov. Krivulja na tej sliki je, tako kot pri grafih na slikah 6.18 in 6.19, zglajena z metodo LOESS [23]. Na osi x sta označeni vrednosti 250 (cca. mediana zahtevkov pri *Jan 2 (20 k)* in *AMF*) ter 1000 (cca. povprečje zahtevkov pri *Jan 2 (20 k)* in *AMF*). Tudi razliko v času med razširitvama *AMF* in *Jan 2 (20 k)* pri uporabi predpomnilnika lahko delno pripišemo drugačnem delovanju le-tega. Spomnimo, da imamo pri meritvah s predpomnilnikom pri vseh primerih v predpomnilniku *vse* filtre, ki jih potrebujemo.



Slika 6.23: Poraba povprečnega časa izvajanja glede na število zahtevkov, potrebnih za izvedbo poizvedbe, ki terja vsebnostne poizvedbe, s tem da razširitvi *AMF* in *Jan 2 (20 k)* uporabljata predpomnilnik za filtre. Skala je linearna.

6.3 Rezultati meritev pri razširitvi z omejitvijo predikatov

V temu razdelku predstavljamo rezultate meritev pri razširitvah *Jan N Filter*. Za referenco smo rezultate primerjali s prvotno različico brez razširitev ter z različicama *AMF* in *Jan 2 (20 k)*.

Predikate, ki smo jih upoštevali, smo pridobili tako, da smo pognali vse poizvedbe z osnovno različico in si ob vsaki vsebnostni poizvedbi shranili njen *predikat*. Nato smo odstranili ponavljajoče predikate in prišli do seznama vseh uporabljenih predikatov v vsebnostnih poizvedbah, ki je viden v tabeli 6.2.

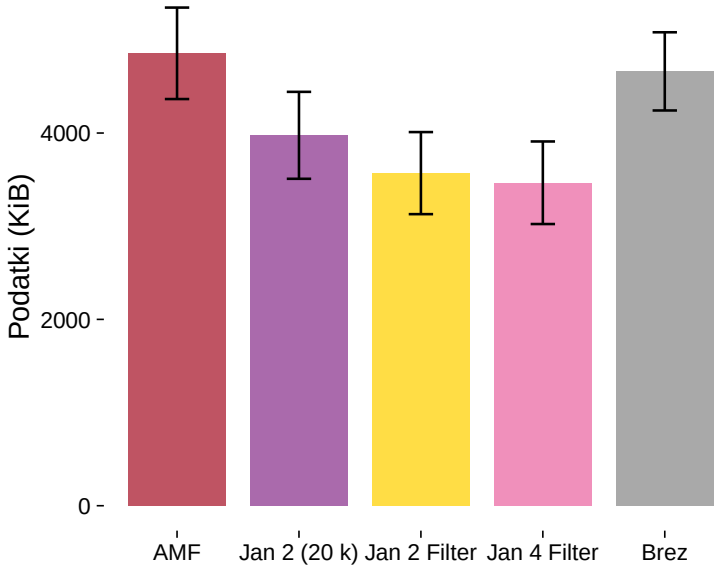
Kot je že omenjeno v razdelku 5.1, protokol za uskladitev seznama v filtru uporabljenih predikatov med strežnikom in odjemalcem ni določen. Omenjen seznam torej pred izvajanjem ročno prekopiramo v kodo strežnika in odjemalca, kot da gre za posebno različico, pri kateri v filtrih upoštevamo samo določene predikate. S preprostim protokolom, kjer bi potrebovali uskladitev enkrat na izvedbo poizvedbe, bi z obravnavanimi podatki in poizvedbami potrebovali okoli 1 KiB dodatnih podatkov na poizvedbo (vsebina tabele 6.2 in režija). Pri naših testnih podatkih in poizvedbah je povprečna količina prenesenih podatkov pri izvajanju poizvedbe z razširitvijo *Jan 2 Filter* okoli 3,5 MiB, s tem da je mediana okoli 1 MiB, torej bi bila razlika sorazmerno majhna.

URI
http://schema.org/language
http://xmlns.com/foaf/age
http://ogp.me/ns#tag
http://db.uwaterloo.ca/~galuc/wsdbm/hasGenre
http://db.uwaterloo.ca/~galuc/wsdbm/likes
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://schema.org/nationality

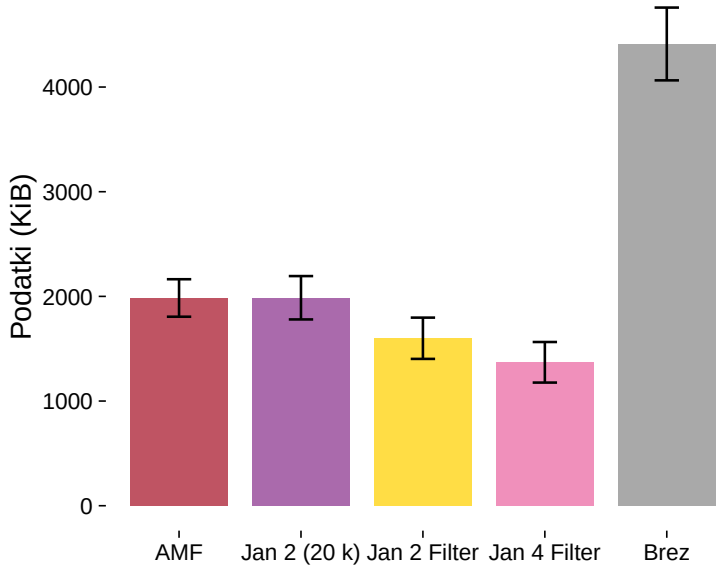
Tabela 6.2: Seznam predikatov v vsebnostnih poizvedbah, ki se zgodijo ob izvedbi *vseh* 340 poizvedb.

6.3.1 Količina prenesenih podatkov

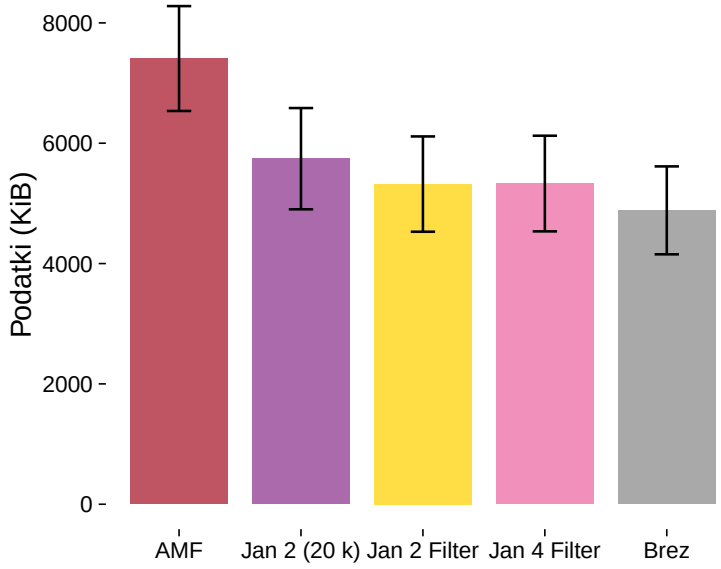
Na sliki 6.24 vidimo primerjavo povprečne količine prenesenih podatkov pri izvedbi *vseh* poizvedb. Vidimo lahko, da različici z razširitvama *Jan 2 Filter* in *Jan 4 Filter* preneseta manj podatkov kot različica z razširitvijo *Jan 2 (20 k)*: $\overline{podatki_{Jan\,2\,Filter}} = 3.569{,}772 < \overline{podatki_{Jan\,2\,(20\,k)}} = 3.975{,}417$, pri čemer je $V = 4.108$ in $p < 2{,}2 \cdot 10^{-16}$; $\overline{podatki_{Jan\,4\,Filter}} = 3.466{,}438 < \overline{podatki_{Jan\,2\,(20\,k)}} = 3.975{,}417$, pri čemer je $V = 1.748$ in $p < 2{,}2 \cdot 10^{-16}$. Če torej vemo, do kakšnih vsebnostnih poizvedb bo prihajalo, lahko optimiramo našo rešitev in pridemo do statistično značilne izboljšave, kar se tiče zmanjšanja prenosa podatkov.



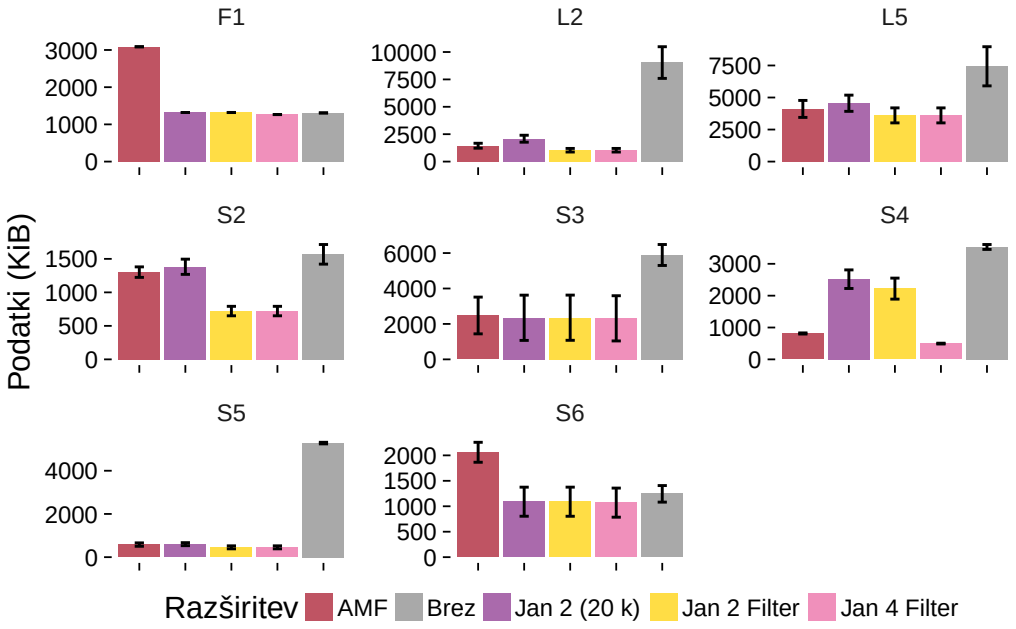
Slika 6.24: Primerjava povprečne količine prenesenih podatkov pri izvedbi *vseh* 340 poizvedb z različnimi razširitvami.



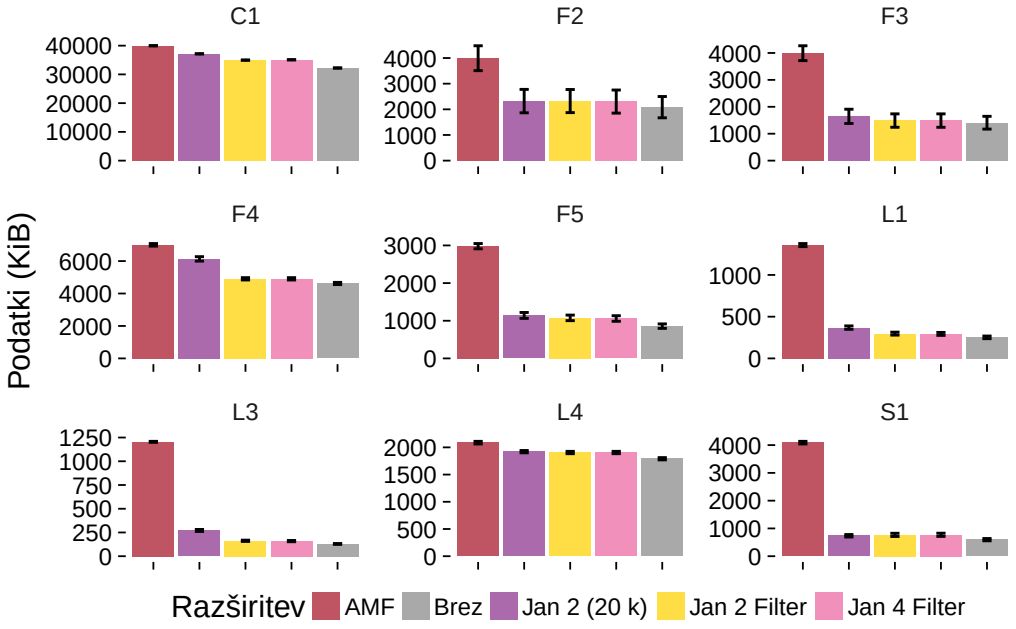
Slika 6.25: Primerjava povprečne količine prenesenih podatkov pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), z različnimi razširitvami.



Slika 6.26: Primerjava povprečne količine prenesenih podatkov pri izvedbi sklopov poizvedb, ki *ne* terjajo vsebnostnih poizvedb (C1, F2, F3, F4, F5, L1, L3, L4, S1), z različnimi razširitvami.



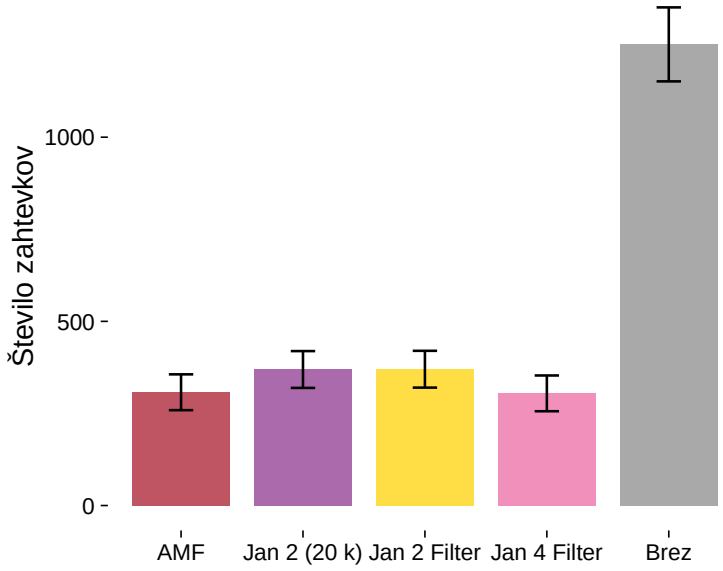
Slika 6.27: Primerjava povprečne količine prenesenih podatkov po sklopih, pri katerih delamo vsebnostne poizvedbe.



Slika 6.28: Primerjava povprečne količine prenesenih podatkov po sklopih, pri katerih *ne* delamo vsebnostnih poizvedb.

6.3.2 Število zahtevkov

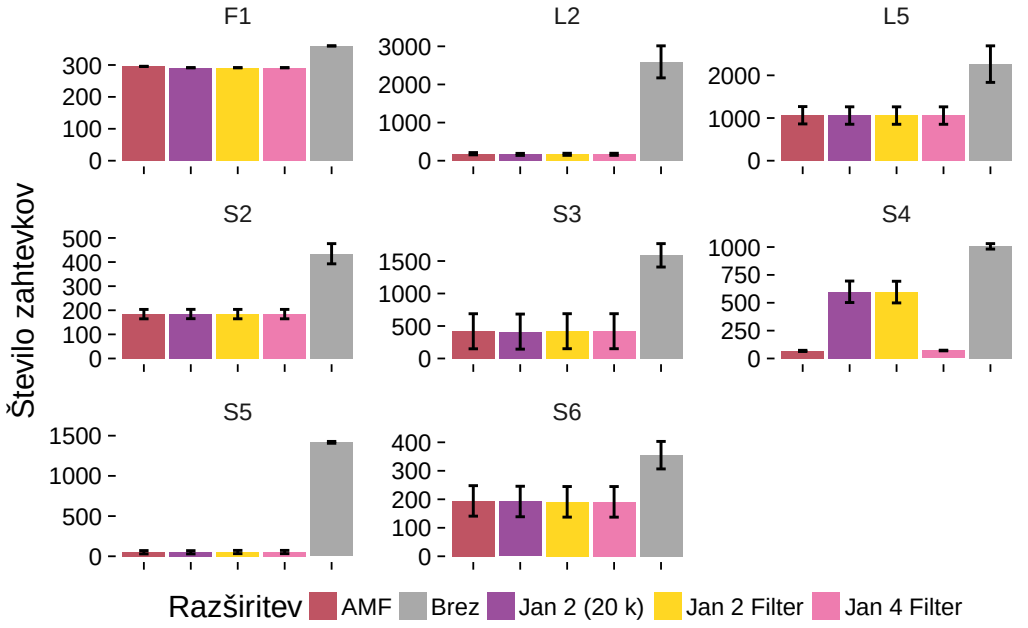
Na sliki 6.29 vidimo primerjavo povprečnega števila zahtevkov pri izvedbi poizvedb, ki terjajo vsebnostne poizvedbe. Obravnavamo samo poizvedbe, ki terjajo vsebnostne poizvedbe, saj pri izvajanju ostalih poizvedb razlike v številu zahtevkov ni. Vidimo lahko, da različica z razširitvijo *Jan 4 Filter* naredi manj zahtevkov kot različica z razširitvijo *Jan 2 (20 k)*, vendar razlika ni statistično značilna: $\overline{zahtevki_{Jan\,4\,Filter}} = 304{,}5312 < \overline{zahtevki_{Jan\,2\,(20\,k)}} = 369{,}2688$, pri čemer je $V = 2.710{,}5$ in $p = 0{,}6106$.



Slika 6.29: Primerjava povprečnega števila zahtevkov pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), z različnimi razširitvami.

Na sliki 6.30 lahko vidimo, da je zmanjšanje števila zahtevkov pri uporabi razširitve *Jan 4 Filter* proti razširitvi *Jan 2 (20 k)* opazno le pri sklopu S4. Če obravnavamo samo število zahtevkov pri izvedbi poizvedb v sklopu S4, je izboljšanje statistično značilno: $\overline{zahtevki_{Jan\,4\,Filter}} = 71{,}6 < \overline{zahtevki_{Jan\,2\,(20\,k)}} = 598{,}6$, pri čemer je $V = 0$ in $p = 3{,}942 \cdot 10^{-5}$. Kot je omenjeno v raz-

delku 6.2.2, je zaradi načina izvajanja poizvedb v tem sklopu glede zmanjšanja števila zahtevkov bolj uspešna razširitev *Jan 4* kot *Jan 2*. S tem, ko v filtru upoštevamo le trojčke, katerih predikat se pojavlja v vsebnostnih poizvedbah, v tem primeru toliko zmanjšamo velikost filtra, da se ga nam splača poslati. To pozitivno vpliva tako na zmanjšanje števila zahtevkov kot na zmanjšanje prenesenih podatkov.

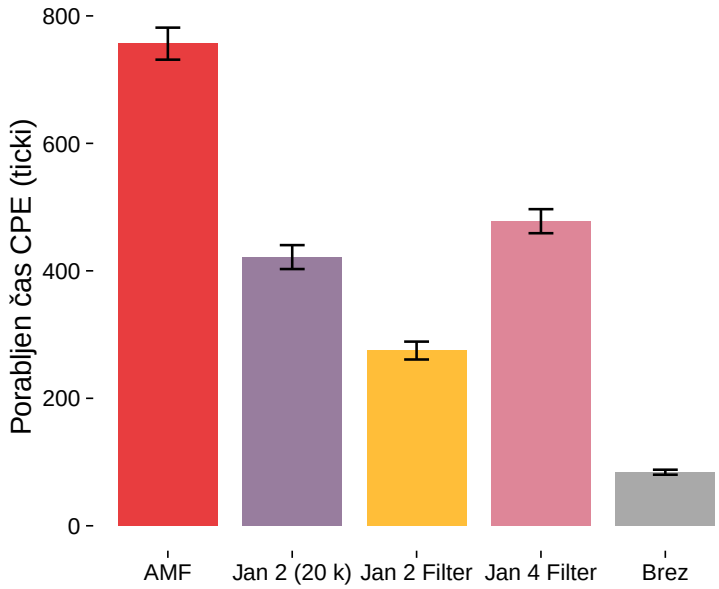


Slika 6.30: Primerjava povprečnega števila zahtevkov po sklopih, pri katerih delamo vsebnostne poizvedbe.

6.3.3 Poraba časa CPE na strežniku

Na sliki 6.31 lahko vidimo, da različica z razširitvijo *Jan 2 Filter* porabi manj časa CPE kot različica z razširitvijo *Jan 2 (20 k)*: $\overline{cpe_{Jan\,2\,Filter}} = 274{,}9404 < \overline{cpe_{Jan\,2\,(20\,k)}} = 421{,}5787$, pri čemer je $V = 27.486$ in $p < 2{,}2 \cdot 10^{-16}$. To je zato, ker filter ob omejitvi predikatov vsebuje manj trojčkov in posledično dela z ustrezno manjšim številom zgoščevalnih funkcij. Pri različici *Jan 4*

Filter je poraba časa CPE večja zaradi načina dostopanja do podatkov; z različico *Jan 4* vsled pridobivanja trojčkov za filter naredimo dvakrat toliko poizvedb do izvora podatkov (v našem primeru je izvor HDT) kot z različico *Jan 2*.



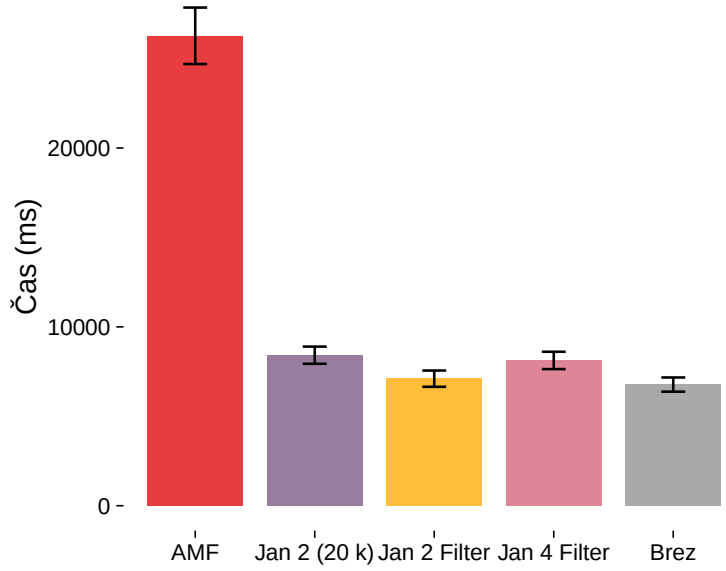
Slika 6.31: Primerjava povprečnega časa CPE pri izvedbi *vseh* 340 poizvedb.

6.3.4 Čas izvajanja poizvedb

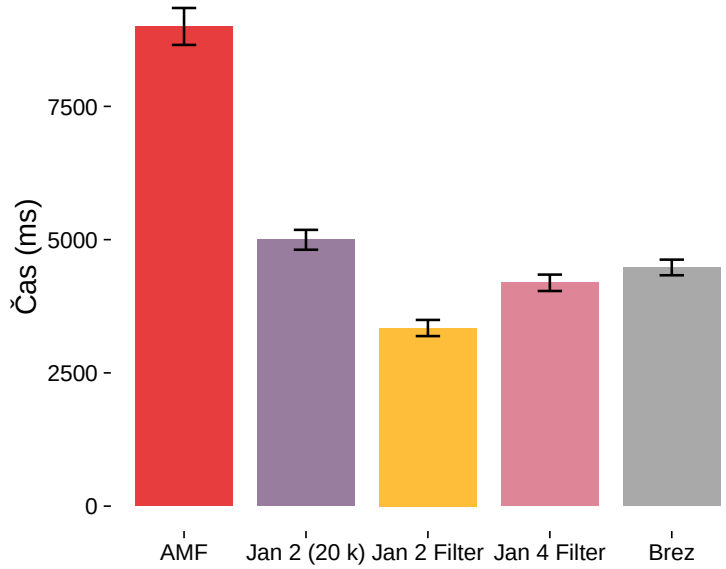
Iz slike 6.32 je razvidno, da razširitev z različico *Jan 2 Filter* pri izvedbi *vseh* poizvedb porabi povprečno manj časa kot razširitev z različico *Jan 2 (20 k)*: $\overline{čas_{Jan\,2\,Filter}} = 7.102{,}792 < \overline{čas_{Jan\,2\,(20\,k)}} = 8.416{,}918$, pri čemer je $V = 43.746$ in $p < 2{,}2 \cdot 10^{-16}$. Razlika je še večja, če primerjamo povprečen porabljen čas pri izvajanju samo tistih poizvedb, ki terjajo vsebnostne poizvedbe (graf na sliki 6.33).

Če uporabimo predpomnilnik za filtre, razlike v povprečnem porabljenem času pri izvedbi *vseh* poizvedb niso več tako očitne (graf na sliki 6.34). Če pri uporabljenem predpomnilniku primerjamo povprečen porabljen čas pri izvedbi samo tistih poizvedb, ki terjajo vsebnostne poizvedbe (graf na

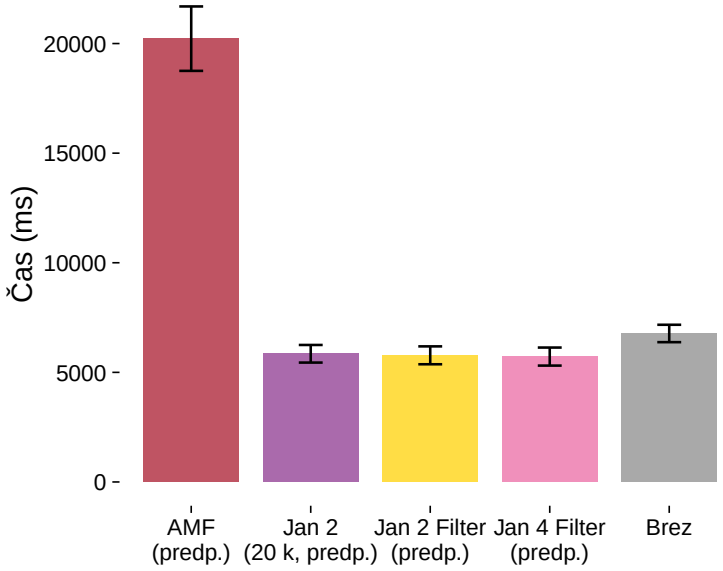
sliki 6.35), je razlika statistično značilna:
$\overline{čas_{Jan\,2\,Filter}} = 2.131{,}052 < \overline{čas_{Jan\,2\,(20\,k)}} = 2.400{,}744$, pri čemer je $V = 40.727$ in $p < 2{,}2 \cdot 10^{-16}$.



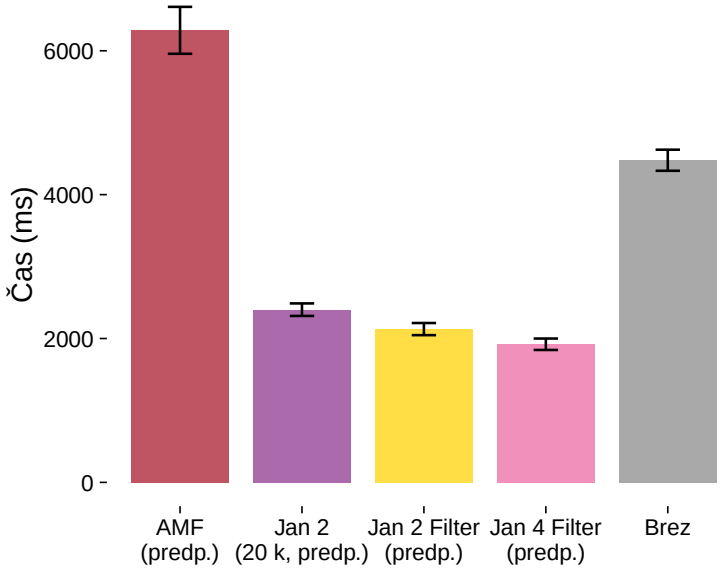
Slika 6.32: Primerjava povprečnega časa izvajanja pri izvedbi *vseh* 340 poizvedb.



Slika 6.33: Primerjava povprečnega časa izvajanja pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6).



Slika 6.34: Primerjava povprečnega časa izvajanja pri izvedbi *vseh* 340 poizvedb, s tem da je uporabljen predpomnilnik za filtre.



Slika 6.35: Primerjava povprečnega časa izvajanja pri izvedbi sklopov poizvedb, ki terjajo vsebnostne poizvedbe (F1, L2, L5, S2, S3, S4, S5, S6), s tem da je uporabljen predpomnilnik za filtre.

Poglavje 7

Grafični vmesnik za sestavljanje in izvajanje poizvedb

Za zaključek celote smo naredili preprost grafični vmesnik, ki omogoča sestavljanje preprostih poizvedb, ki imajo en BGP, torej ne podpira sestavljanja poizvedb s funkcijami, kot je npr. unija BGP-jev. Kljub temu lahko s pomočjo omenjenega vmesnika sestavimo vse poizvedbe, opisane v razdelku 6.1, ki smo jih uporabljali za meritve. Vmesnik deluje tako, da sestavljamo BGP z vnašanjem vzorcev trojčkov, pri čemer nam nudi pomoč v obliki samodejnega dopolnjevanja (angl. *autocomplete*). Izbirniki vzorcev trojčkov delujejo neodvisno med sabo, z izjemo tega, da nudijo samodejno dopolnitev spremenljivk, uporabljenih v ostalih vzorcih. Vmesnik nam prav tako filtrira predlagane URI-je glede na že vpisane URI-je, torej če npr. za *predikat* vzamemo `tags:taggedWithTag`, dobimo kot predloge za *objekt* le oznake albumov, kot predloge za *subjekt* pa albume. Če vpišemo dva URI-ja, lahko dobimo (poleg spremenljivk) največ en predlog za dopolnitev, ki ustreza trojčku v bazi. Grafični vmesnik kot vir podatkov uporablja relacijsko bazo MariaDB, ki je lahko tudi, kot je opisano v poglavju 3.2, uporabljena kot vir podatkov za strežnik delcev vzorcev trojčkov. Strežniška aplikacija grafičnega vmesnika je realizirana v Node.js [8] in uporablja knjižnico Express [4], na strani odjemalca pa smo uporabili knjižnici Select2 [10] in

jQuery [5]. Programska koda grafičnega vmesnika je dostopna v repozitoriju GitHub [34].

Pri izvedbi poizvedb SQL, kjer iščemo ujemanje dela niza, ki je lahko bodisi na začetku bodisi na koncu, npr. pri poizvedbi 7.1, moramo preiskati celo tabelo. Ker gre za veliko količino trojčkov, tako poizvedovanje ni primerno, saj lahko terja veliko časa. Način indeksiranja, ki omogoča optimirano iskanje besed v povedih, je opisan v članku [21]. Lahko bi prilagodili omenjen način za iskanje po delih URI-jev, vendar smo se odločili za preprostejši način. Ker poizvedbe, kjer nas zanima ujemanje začetnega dela niza (kot poizvedba 7.2), ob prisotnosti indeksa ne zahtevajo preiskovanja cele tabele, smo se odločili, da poleg samih URI-jev hranimo še zadnje dele URI-jev (del URI-ja po zadnji poševnici). Način zapisa podatkov je viden v tabeli 7.1. Vmesnik deluje tako, da dopolnjuje tako začetke zadnjih delov URI-jev kot začetke celih URI-jev. V primeru literalov je del URI-ja prazen in vmesnik dopolnjuje le celoten literal.

```
1 SELECT * FROM trojcki WHERE p LIKE "%tag%";
```

Poizvedba 7.1: Poizvedba SQL, za izvedbo katere *je* potrebno preiskovanje cele tabele.

```
1 SELECT * FROM trojcki WHERE p_del LIKE "tag%";
```

Poizvedba 7.2: Poizvedba SQL, za izvedbo katere ob prisotnosti indeksa *ni* potrebno preiskovanje cele tabele.

s	p	o	s_del	p_del	o_del
record:10011	tags:taggedWithTag	tag:pop	10011	taggedWithTag	pop
record:1003	tags:taggedWithTag	tag:pop	1003	taggedWithTag	pop
record:1003	tags:taggedWithTag	tag:rock	1003	taggedWithTag	rock
record:10031	tags:taggedWithTag	tag:rock	10031	taggedWithTag	rock
record:10034	tags:taggedWithTag	tag:jazz	10034	taggedWithTag	jazz
record:1006	tags:taggedWithTag	tag:metal	1006	taggedWithTag	metal

Tabela 7.1: Izsek podatkov iz relacijske baze z dodanimi polji `s_del`, `p_del` in `o_del`, ki predstavljajo zadnje dele URI-jev `s`, `p` in `o`. Zaradi berljivosti so URI-ji v tabeli skrajšani z istimi predponami kot pri poizvedbi 3.2.

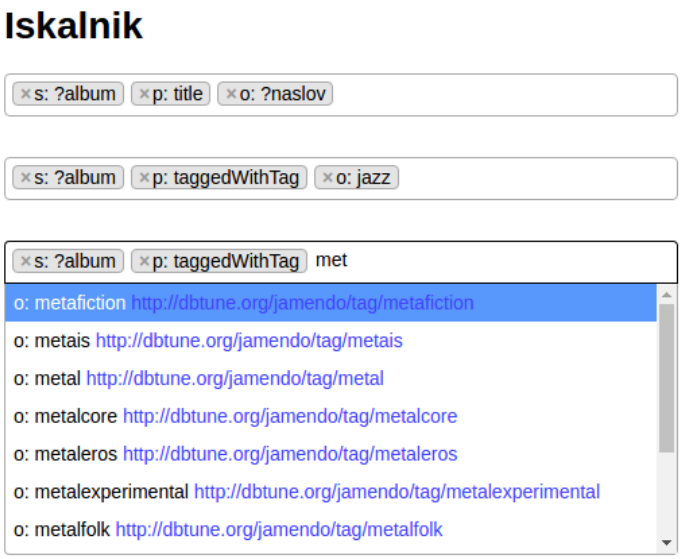
Omenjen grafični vmesnik omogoča tudi izvajanje sestavljenih poizvedb iz brskalnika s pomočjo odjemalca delcev vzorcev trojčkov. Uporabimo lahko katerokoli od v tem delu navedenih različic odjemalcev. Pri tem smo uporabili pripomoček "Linked Data Fragments jQuery Widget" [43], ki smo ga rahlo predelali, zato da smo lahko izpisali rezultate v želeni obliki.

Na sliki 7.1 je viden grafični vmesnik pri izboru *objekta* vzorca trojčka. Ker je *subjekt* spremenljivka, *predikat* pa je `tags:taggedWithTag`, imamo za izbiro, poleg poljubnih spremenljivk, na voljo *objekte*, ki nastopajo v trojčkih s *predikatom* `tags:taggedWithTag`. Ko izberemo želene vzorce trojčkov, s klikom na gumb "Išči" zaženemo iskanje, rezultati pa se začnejo sproti ob izvajanju poizvedbe pojavljati na zaslonu, kot prikazuje slika 7.2. V primeru vzorcev trojčkov, ki so izbrani na sliki 7.2, se izvede poizvedba SPARQL, podobna poizvedbi 7.3.

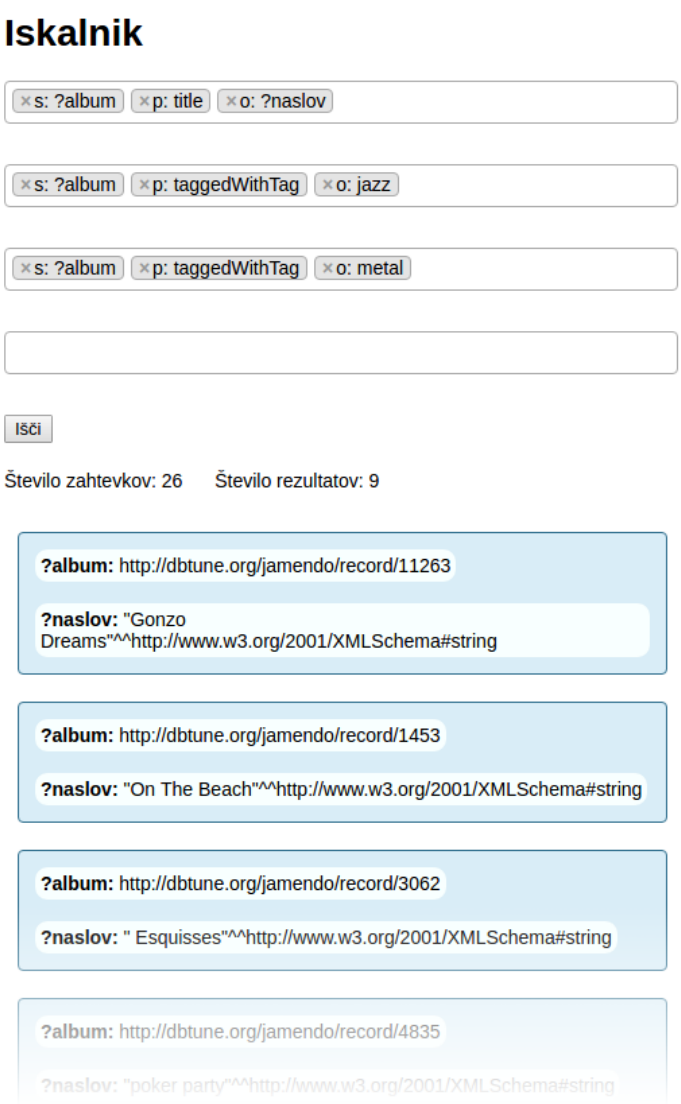
```
PREFIX tags:   <http://www.holygoat.co.uk/owl/redwood/0.1/tags/>
PREFIX tag:    <http://dbtune.org/jamendo/tag/>
PREFIX record: <http://dbtune.org/jamendo/record/>
PREFIX dc:     <http://purl.org/dc/documents/1.1/>

SELECT ?album ?naslov WHERE {
    ?album dc:title ?naslov.
    ?album tags:taggedWithTag tag:jazz.
    ?album tags:taggedWithTag tag:metal.
}
```

Poizvedba 7.3: Poizvedba SPARQL, ki jo sestavi grafični vmesnik ob upoštevanju izbire na sliki 7.2.



Slika 7.1: Dopolnjevanje trojčka pri sestavljanju poizvedbe.



Slika 7.2: Prikaz razultatov po izvedbi sestavljene poizvedbe.

Poglavje 8

Zaključek

8.1 Rezultati našega dela

V našem delu smo naredili nekaj različnih prispevkov, ki jih povzamemo s sledečim seznamom.

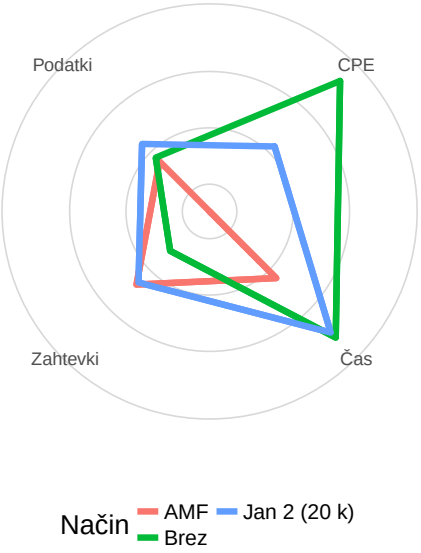
Popravek razširitve AMF. Kot je omenjeno v razdelku 4.2, smo rešili problem implementacije razširitve *AMF* in s tem, sodeč po rezultatih naših testov, precej izboljšali njeno delovanje. Naš popravek je bil kmalu vključen v javni repozitorij in ga v času pisanja tega dela vsebuje aktualna različica razširitve.

Naše razširitve. Glavni prispevek našega dela je družina razširitev, ki v metapodatke delcev vzorcev namesto podatkov o celotnem delcu (kot dela razširitev *AMF*) doda podatke, povezane z deli trojčkov (z *subjekti* oziroma *objekti*) na prikazani strani delca. S tem smo zmanjšali število zahtevkov pri izvedbi poizvedb, ki terjajo vsebnostne poizvedbe in, posledično, količino prenesenih podatkov.

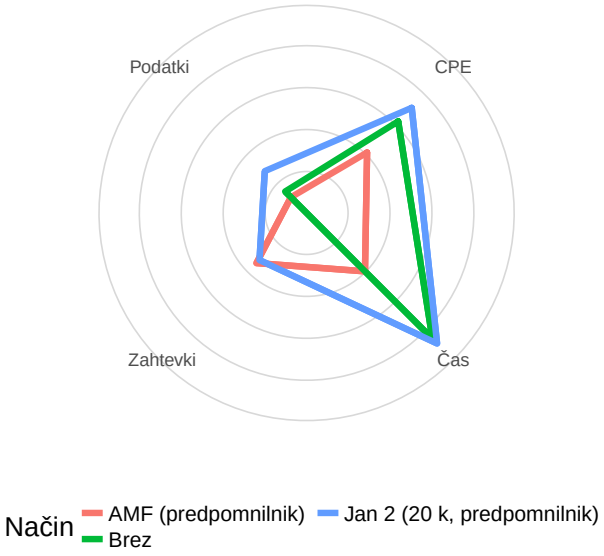
Vmesnik, ki črpa podatke iz relacijske baze. Naredili smo vmesnik, ki omogoča, da strežnik delcev vzorcev trojčkov črpa podatke (trojčke) iz relacijske baze MariaDB.

Preprost grafični vmesnik. Za zaključek celote smo naredili grafičen vmesnik, ki omogoča samodejno dopolnjevanje trojčkov in z njimi sestavljanje preprostih poizvedb SPARQL ter njihovo izvajanje.

Na slikah 8.1 in 8.2 vidimo primerjavo osnovne različice z razširitvama *Jan 2 (20 k)* in *AMF* brez uporabe in z uporabo predpomnilnika za filtre. Pri teh grafih so zaželene večje oblike, saj večja vrednost pri določeni metriki v tem primeru pomeni manjšo porabo vira, torej ravno obratno kot pri poglavju 6. Vsaka metrika posebej je bila, neodvisno od drugih metrik, normalizirana na interval [0,100], tako da je največja vrednost vsake posamezne metrike 100, nato pa smo vsako metriko pomnožili z -1 in dodali 100. Ker ima osnovna različica brez razširitev visoko vrednost pri metriki CPE, je torej dobra v smislu majhne porabe časa CPE. Pri meritvah časa se moramo zavedati, da bi bil v realnem scenariju čas bolj odvisen od števila zahtevkov, saj smo pri naših meritvah imeli strežnik in odjemalec na istem računalniku, torej so bili odgovori na zahteve hitro preneseni. Iz grafov je razvidno, da je, glede na test, ki smo ga izvedli, naša metoda uspešna. Na tem mestu velja ponovno omeniti, da lahko razliko v porabi časa CPE med razširitvama *Jan 2 (20 k)* in *AMF* pri uporabi predpomnilnika za filtre pripišemo drugačni implementaciji le-tega.



Slika 8.1: Mrežni graf, ki primerja osnovno različico z razširitvama *Jan 2 (20 k)* in *AMF*. Pri tem grafu večja vrednost pomeni manjšo porabo vira, torej so zaželene večje oblike.



Slika 8.2: Mrežni graf, ki primerja osnovno različico z razširitvama *Jan 2 (20 k)* in *AMF*, pri čemer slednji uporabljata predpomnilnik za filtre. Pri tem grafu večja vrednost pomeni manjšo porabo vira, torej so zaželene večje oblike.

8.2 Nadaljnje delo

V našem delu smo pri vseh meritvah uporabljali isto verjetnost napake in isto velikost strani. Lahko bi se odločili za optimizacijo teh parametrov, ki imajo verjetno velik vpliv na delovanje naše razširitve. Pri naši razširitvi, za razliko od razširive *AMF*, namreč velikost strani vpliva na vsebino Bloomovega filtra v metapodatkih.

Zanimiva bi bila združitev naše razširitve z optimizacijo, opisano v članku [36]. Pri tej optimizaciji se namesto preprostega izvajanja vsebnostnih poizvedb algoritem odloči, ali je namesto izvajanja dejanskih vsebnostnih poizvedb bolj smiselno prenesti celotne delce vzorcev trojčkov, ki so potrebni za preverjanje danih vsebnosti. Slaba stran omenjene optimizacije je, da je na odjemalcu za to potrebna lokalna hramba trojčkov, kar lahko, predvsem za naprave z omejenim pomnilnikom, povzroča težave. V primeru prenosa celotnih delcev z našo razširitvijo ne dosežemo izboljšav, ampak zaradi dodanih metapodatkov kvečjemu povečamo količino prenesenih podatkov. Algoritem z omenjeno optimizacijo se odloči za izvajanje vsebnostnih poizvedb takrat, ko so delci vzorcev tročjkov, ki bi jih morali prenesti, preveliki, da bi se splačal njihov prenos. V slednjem primeru lahko, kar se tiče števila zahtevkov in posledično količine prenesenih podatkov, različica z našo razširitvijo doseže veliko boljše rezultate kot osnovna različica. Vsekakor pa je izboljšava odvisna tako od podatkov kot od poizvedb, ki jih nad njimi izvajamo.

V naši razširitvi smo uporabljali Bloomove filtre. Lahko bi se odločili za kompresijo le-teh, o kateri med drugim govori članek [30].

Pri *Jan N Filter*, kot je omenjeno v poglavju 5, nismo določili protokola za uskladitev seznama v filtru uporabljenih predikatov. Po opredelitvi omenjenega protokola bi bilo zanimivo, če bi strežnik uporabljene predikate sproti prilagajal glede na zahtevane vsebnostne poizvedbe. V tem primeru bi lahko imeli praktično uporabno rešitev, ki se približa dobrim rezultatom, ki smo jih dosegli z razširitvijo *Jan 4 Filter*.

8.3 Sklep

Kot smo že omenili na začetku, uporabo povezanih podatkov zavirajo problemi s performancami poizvedovanja in posledično z dostopnostjo oddaljenih točk SPARQL. V tem delu smo se osredotočili na izboljšavo v smislu zmanjšanja števila vsebnostnih poizvedb in zmanjšanja količine prenesenih podatkov pri izvajanju poizvedb SPARQL nad delci vzorcev trojčkov. Naša razširitev se glede na naš test, kot lahko razberemo iz grafov na slikah 8.1 in 8.2, izkaže za uspešno. Glede na pridobljene rezultate menimo, da ima naša razširitev praktično uporabnost. Delce vzorcev trojčkov z našo razširitvijo bi lahko npr. uporabili kot spletni programski vmesnik pri družbenem omrežju, pri čemer bi izbrali različico *Jan 4 Filter*, če bi večina vsebnostnih poizvedb vsebovala le del predikatov (kot npr. `je-prijatelj-od`, `je-všeč`).

Za zaključek velja omeniti, da je gonilo napredka interneta, kot tudi omenja članek [24], njegova odprtost in nevtralnost, zaradi različnih interesov pa sta ravno ti lastnosti pogosto pod vprašanjem. V našem delu se sicer ne ukvarjamo neposredno s tem, vendar pa celotna vsebina našega dela stoji na teh načelih. Internet je nedvomno postal del našega življenja in nam ponuja veliko priložnosti, hkrati pa je tarča vseh, ki bi ga radi prilagodili v prid svojim interesom.

Literatura

[1] "BrowserMob Proxy", https://github.com/lightbody/browsermob-proxy. Dostopano: 6. 10. 2016.

[2] "DBTune - Jamendo RDF dump", http://dbtune.org/jamendo/. Dostopano: 6. 10. 2016.

[3] "Example of a Bloom filter", https://commons.wikimedia.org/wiki/File:Bloom_filter.svg. Dostopano: 6. 10. 2016.

[4] "Express", http://expressjs.com/. Dostopano: 6. 10. 2016.

[5] "jQuery", https://jquery.com/. Dostopano: 6. 10. 2016.

[6] "Linked Data", https://www.w3.org/DesignIssues/LinkedData.html. Dostopano: 6. 10. 2016.

[7] "Linux man-pages project. proc(5) - Linux manual page", http://man7.org/linux/man-pages/man5/proc.5.html. Dostopano: 6. 10. 2016.

[8] "Node.js", https://nodejs.org/. Dostopano: 6. 10. 2016.

[9] "Pošta Slovenije", http://www.posta.si/. Dostopano: 6. 10. 2016.

[10] "Select2", https://select2.github.io/. Dostopano: 6. 10. 2016.

[11] "Selenium WebDriver", http://www.seleniumhq.org/. Dostopano: 6. 10. 2016.

[12] G. Aluç, O. Hartig, M. T. Özsu, K. Daudjee, "Diversified Stress Testing of RDF Data Management Systems", v *International Semantic Web Conference*. Springer, 2014, str. 197–212.

[13] C. B. Aranda, A. Hogan, J. Umbrich, P.-Y. Vandenbussche, "SPARQL Web-Querying Infrastructure: Ready for Action?", v *The 12th International Semantic Web Conference (ISWC2013)*, 2013, str. 273–288.

[14] C. Bizer, T. Heath, T. Berners-Lee, "Linked Data - The Story So Far", zv. 5, št. 3, 2009, str. 1–22.

[15] C. Bizer, R. Cyganiak, T. Gauß, "The RDF Book Mashup: From Web APIs to a Web of Data", v *The 3rd Workshop on Scripting for the Semantic Web (SFSW 2007), Innsbruck, Austria*, 2007.

[16] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors", *Communications of the ACM*, zv. 13, št. 7, str. 422–426, 1970.

[17] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, G. Varghese, "An Improved Construction for Counting Bloom Filters", v *European Symposium on Algorithms*. Springer, 2006, str. 684–695.

[18] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, B. Bhattacharjee, "Building an Efficient RDF Store Over a Relational Database", v *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, str. 121–132.

[19] A. Broder, M. Mitzenmacher, "Network applications of Bloom filters: A survey", *Internet mathematics*, zv. 1, št. 4, str. 485–509, 2004.

[20] G. Carothers, E. Prud'hommeaux, "RDF 1.1 Turtle", W3C, W3C Recommendation, feb. 2014, http://www.w3.org/TR/2014/REC-turtle-20140225/.

[21] S. Dessloch, N. Mattos, "Integrating SQL Databases with Content-Specific Search Engines", v *VLDB*, zv. 97, 1997, str. 528–537.

[22] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, "Hypertext Transfer Protocol", Internet Requests for Comments, RFC 2616, jun. 1999.

[23] J. Fox, S. Weisberg, *An R companion to applied regression*. Sage, 2010.

[24] P. Ganley, B. Allgrove, "Net neutrality: A user's guide", *Computer Law & Security Review*, zv. 22, št. 6, str. 454–463, 2006.

[25] S. Harris, N. Shadbolt, "SPARQL Query Processing with Conventional Relational Database Systems", v *International Conference on Web Information Systems Engineering*. Springer, 2005, str. 235–244.

[26] M. Hausenblas, W. Halb, Y. Raimond, T. Heath, "What is the Size of the Semantic Web", *Proceedings of I-Semantics*, str. 9–16, 2008.

[27] T. Heath, C. Bizer, "Linked Data: Evolving the Web into a Global Data Space", *Synthesis Lectures on the Semantic Web: Theory and Technology*, zv. 1, št. 1, str. 1–136, 2011.

[28] E. Hervalejo, M. A. Martínez-Prieto, J. D. Fernández, O. Corcho, "HDTourist: exploring urban data on android", v *Proceedings of the 2014 International Conference on Posters & Demonstrations Track-Volume 1272*. CEUR-WS.org, 2014, str. 65–68.

[29] B. M. Maggs, R. K. Sitaraman, "Algorithmic Nuggets in Content Delivery", *ACM SIGCOMM Computer Communication Review*, zv. 45, št. 3, str. 52–66, 2015.

[30] M. Mitzenmacher, "Compressed Bloom Filters", *IEEE/ACM Transactions on Networking (TON)*, zv. 10, št. 5, str. 604–612, 2002.

[31] J. Odvarko, A. Jain, A. Davies, "HTTP Archive (HAR) format", *W3C draft*, 2012.

[32] C. Pautasso, O. Zimmermann, F. Leymann, "RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision", v *Proceedings of the 17th international conference on World Wide Web*. ACM, 2008, str. 805–814.

[33] F. Putze, P. Sanders, J. Singler, "Cache-, Hash- and Space-Efficient Bloom Filters", v *International Workshop on Experimental and Efficient Algorithms*. Springer, 2007, str. 108–121.

[34] J. Robas, "TpfAutocomplete: Grafični vmesnik za sestavljanje in izvajanje poizvedb SPARQL", https://github.com/janrobas/TpfAutocomplete. Dostopano: 6. 10. 2016.

[35] A. Rodriguez, "RESTful Web services: The basics", *IBM developerWorks*, 2008.

[36] J. Van Herwegen, R. Verborgh, E. Mannens, R. Van de Walle, "Query Execution Optimization for Clients of Triple Pattern Fragments", v *The Semantic Web. Latest Advances and New Domains*, ser. Lecture Notes in Computer Science.

[37] M. Vander Sande, R. Verborgh, J. Van Herwegen, "Linked Data Fragments Client", https://github.com/LinkedDataFragments/Client.js. Dostopano: 6. 10. 2016.

[38] M. Vander Sande, R. Verborgh, J. Van Herwegen, "Linked Data Fragments client (AMF)", https://github.com/LinkedDataFragments/Client.js/tree/amq. Dostopano: 6. 10. 2016.

[39] M. Vander Sande, R. Verborgh, J. Van Herwegen, "Linked Data Fragments Server", https://github.com/LinkedDataFragments/Server.js. Dostopano: 6. 10. 2016.

[40] M. Vander Sande, R. Verborgh, J. Van Herwegen, E. Mannens, R. Van de Walle, "Opportunistic Linked Data Querying through Approximate Membership Metadata", v *The Semantic Web – ISWC 2015*.

[41] M. Vander Sande, R. Verborgh, J. Van Herwegen, J. Robas, "TriplePatternFragmentsJan: Linked Data Fragments strežnik in odjemalec z našo razširitvijo", https://github.com/janrobas/TriplePatternFragmentsJan. Dostopano: 6. 10. 2016.

[42] R. Verborgh, "DBpedia's Triple Pattern Fragments: Usage Patterns and Insights", v *The Semantic Web: ESWC 2015 Satellite Events*.

[43] R. Verborgh, "Linked Data Fragments jQuery Widget", https://github.com/LinkedDataFragments/jQuery-Widget.js. Dostopano: 6. 10. 2016.

[44] R. Verborgh, O. Hartig, B. De Meester, G. Haesendonck, L. De Vocht, M. Vander Sande, R. Cyganiak, P. Colpaert, E. Mannens, R. Van de Walle, "Querying Datasets on the Web with High Availability", v *Proceedings of the 13th International Semantic Web Conference*, ser. Lecture Notes in Computer Science, zv. 8796, okt. 2014, str. 180–196.

[45] R. Verborgh, M. Vander Sande, O. Hartig, J. Van Herwegen, L. De Vocht, B. De Meester, G. Haesendonck, P. Colpaert, "Triple Pattern Fragments: a Low-cost Knowledge Graph Interface for the Web", *Journal of Web Semantics*, zv. 37–38, str. 184–206, 2016.